Studies in Computational Intelligence 269

Francisco Fernández de Vega
Erick Cantú-Paz (Eds.)

Parallel and Distributed Computational Intelligence

Springer

Francisco Fernández de Vega and Erick Cantú-Paz (Eds.)

Parallel and Distributed Computational Intelligence

Studies in Computational Intelligence, Volume 269

Editor-in-Chief
Prof. Janusz Kacprzyk
Systems Research Institute
Polish Academy of Sciences
ul. Newelska 6
01-447 Warsaw
Poland
E-mail: kacprzyk@ibspan.waw.pl

Francisco Fernández de Vega and
Erick Cantú-Paz (Eds.)

Parallel and Distributed Computational Intelligence

Springer

Francisco Fernández de Vega
University of Extremadura
C/ Sta Teresa de Jornet, 38
Merida - Spain
E-mail: fcofdez@unex.es

Erick Cantu-Paz
Yahoo! Inc.
701 First Avenue
Sunnyvale, CA 94087
USA
E-mail: cantupaz@acm.org

ISBN 978-3-642-10674-3 e-ISBN 978-3-642-10675-0

DOI 10.1007/978-3-642-10675-0

Studies in Computational Intelligence ISSN 1860-949X

Library of Congress Control Number: 2009943374

Typeset & *Cover Design:* Scientific Publishing Services Pvt. Ltd., Chennai, India.

Printed in acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Contents

Introduction

Francisco Fernández de Vega and Erick Cantú-Paz

The growing success of biologically inspired algorithms in solving large and complex problems has spawned many interesting areas of research. Over the years, one of the mainstays in bio-inspired research has been the exploitation of parallel and distributed environments to speedup computations and to enrich the algorithms. From the early days of research on bio-inspired algorithms, their inherently parallel nature was recognized and different parallelization approaches have been explored. Parallel algorithms promise reductions in execution time and open the door to solve increasingly larger problems. But parallel platforms also inspire new bio-inspired parallel algorithms that, while similar to their sequential counterparts, explore search spaces differently and offer improvements in solution quality.

Our objective in editing this book was to assemble a sample of the best work in parallel and distributed biologically inspired algorithms. We invited researchers in different domains to submit their work. We aimed to include diverse topics to appeal to a wide audience. Some of the chapters summarize work that has been ongoing for several years, while others describe more recent exploratory work. Collectively, these works offer a global snapshot of the most recent efforts of bioinspired algorithms' researchers aiming at profiting from parallel and distributed computer architectures—including GPUs, Clusters, Grids, volunteer computing and p2p networks as well as multi-core processors.

We hope this volume will be of value to a wide set of readers, including, but not limited to specialists in Bioinspired Algorithms, Parallel and Distributed Computing, as well as computer science students trying to figure out new paths towards the future of computational intelligence.

This book is a collective effort, and we must thank all the contributing authors, whose effort and dedication have given rise to the present work. Last but not least we appreciate the encouragment, support and patience offered by Professor Janusz Kacprzyk, as well as by Springer during the editing process.

1 Road Map

The chapters included in this book reflect the diversity of the research activity in parallel and distributed bio-inspired algorithms. We attempted to organize the book around three different themes: platforms, algorithms, and applications. We do not explicitly separate the chapters into sections, because most of the chapters can be categorized into more than one group.

On the chapters that describe different platforms, we have descriptions of implementations of bio-inspired algorithms on very diverse computing resources such as computational grids, graphic processing units, volunteer networks, peer-to-peer systems, and graphic processing units. The next group of chapters describe diverse algorithms. We begin with a review of parallel estimation of distribution algorithms, followed by two chapters on different multi-objective optimization algorithms, and a chapter on efficient update strategies for multi-agent simulations. The final group of chapters includes applications in machine learning, embedded system optimization, protein structure prediction, and modeling of laser dynamics.

The reminder of this section has brief summaries of each chapter.

Chapter 1. *When Huge is Routine: Scaling Genetic Algorithms and Estimation of Distribution Algorithms via Data-Intensive Computing* by Xavier Llorà, Abhishek Verma, Roy H. Campbell, and David E. Goldberg.

Recent data-intensive computing platforms such as Google's MapReduce and Yahoo!'s open source Hadoop enable the routine processing of petabytes of data. In their chapter, Llorà, Verma, Campbell and Goldberg show how data-intensive computing platforms can greatly benefit the parallelization of evolutionary algorithms. The authors explore the use of Hadoop and Meandre, a tool developed in the National Center for Supercomputing Applications of the University of Illinois at Urbana-Champaign. To illustrate the promise of these platforms to speedup evolutionary algorithms, the authors implemented representative EAs that span the range from classic algorithms to estimation of distribution algorithms: a selectorecombinative genetic algorithm, a compact GA, and the extended compact GA.

Llorà et al. present details of their implementations that preserve the behavior of the three algorithms and show experimental results that suggest that the evolutionary algorithms scale well on both platforms. In fact, linear speedups are limited only by the availability of resources.

Chapter 2. *Evolvable Agents: a framework for Peer-to-Peer Evolutionary Algorithms* by J.L.J. Laredo and J.J. Merelo and P.A. Castillo.

In this chapter, Laredo, Merelo, and Castillo present an overview of the properties, performance and issues of EAs implemented on peer-to-peer (P2P) systems. On P2P systems, the computational resources are provided by a group of users who share their spare CPU cycles via the Internet. The authors present a way to parallelize EAs on P2P systems based on an Evolvable Agent model, in which a population of agents perform selection, variation and fitness evaluation. The authors describe their newscast algorithm that defines the communication structure among the evolvable agents. The main issues for the parallelization of EAs on P2P systems are decentralization, scalability and fault tolerance, and the authors describe their approaches to each of these challenges. In terms of decentralization, the proposed algorithm is fully distributed and there is no central control, so but its very nature, the algorithm is not centralized. The authors study the scalability of the algorithm with experiments using trap functions and show that their method scales better to the problem size than a canonical GA. The fault tolerance of the algorithm is demonstrated with a series of experiments where the churn of nodes, expressed as the session length of the participating peers, is modeled with a Weibull distribution. The authors also present some results on dynamic optimization problems.

Chapter 3. *Evolutionary Algorithms on Volunteer Computing Platforms: The MilkyWay@Home Project* by Nate Cole, Travis Desell, Daniel Lombraña González, Francisco Fernández de Vega, Malik Magdon-Ismail, Heidi Newberg, Boleslaw Szymanski, and Carlos Varela.

Volunteer computing grids are composed of Internet-connected computers volunteered by users worldwide. They present a huge potential of computing power that can be harnessed to solve very high-scale problems. In their chapter Cole et al. describe their project to harness volunteer computing to find substructure in the Milky Way galaxy using evolutionary algorithms, the MilkyWay@Home project.

The authors describe the BOINC platform, a volunteer computing technology that is an offshoot of the celebrated SETI@Home project. In a nutshell, BOINC consists of a server that hosts the experiments and creates “work units” and clients that run on the volunteer machines and execute the work. For running evolutionary algorithms on volunteer grids, there are two basic approaches: distribute fitness evaluations or execute experiments in parallel (for example to sweep different parameter settings). In the application described in this chapter, the approach is to distribute fitness evaluations to the volunteers.

The authors compare their volunteer approach using 1000 volunteered computers to computations on 1024 nodes on a BlueGene/L computer and note that, while the approach used on the volunteer computers may take more iterations, it is more accessible and more experiments can be performed, as the computing resources are not shared among many researchers as the BlueGene/L. In addition, the volunteer framework can perform numerous searches at a fraction of the cost of dedicated resources. These results are very promising for the use of volunteer computing for computationally intensive scientific modeling.

Chapter 4. *Self-coordinated on-chip parallel computing: a Swarm Intelligence approach* by Danilo Pani and Luigi Raffo.

Multicores are becoming prevalent, but suffer from being difficult to program. The authors claim that new ideas are needed to support the development of massively parallel architectures. One of these ideas is to search for inspiration in natural systems.

Pani and Raffo define Swarm Intelligence (SI) as "a bioinspired paradigm that takes inspiration from natural swarms, large sets of simple individuals with limited capabilities able to carry out complex tasks exploiting cooperation and self organization." They pioneered the use of SI for the design of digital architectures. This chapter recounts their work from their first explorations to their latest results: a coprocessing unit for fixed point array processing. Their approach shows significant performance improvements without any programming effort and without complex tools for compilation and mapping.

Along the way, Pani and Raffo identify the limitations of their experiments and show how they overcame those limits.

Chapter 5. *Large Scale Bioinformatics Data Mining with Parallel Genetic Programming on Graphics Processing Units* by William B. Langdon.

Graphic Processing Units (GPUs) promise teraflop computing in a desktop for a few hundred US dollars. Langdon gives an overview of the use of GPUs in bioinformatics and computational intelligence. He describes a successful application of genetic programming on GPUs to a Bioinformatics to find a small number of indicative mRNA gene transcript signals from breast cancer tissue samples.

GPUs provide a restricted type of parallel processing, in which each of the many processors simultaneously runs the same program on different data items. This is often referred to a single instruction multiple data (SIMD) or more precisely single program multiple data (SPMD). In the case of Langdon's application, the single instruction belongs to the interpreter and the multiple data are the multiple GP trees. The single interpreter is loaded onto every stream processor within the GPU, and so in every clock tick, the GPU can interpret a part of 128 different GP trees.

The data mining system described in the chapter works in two steps. The first step is used to identify approximately 100 000 inputs to chose from to create a classifier in the second step. The first step has a population of five million trees laid out on a rectangular grid divided into 256x256 squares. At the end of the run, the genetic composition of the best individual in each square was recorded and the inputs that appeared frequently were used in the second step. The final result is a non-linear classifier with only three inputs that has good accuracy on predicting the patient's outcome 10+ years in the future. On this application, Langdon achieved a speedup of more 7x over a program run on the same CPU used to host the GPU.

Chapter 6. *A Review on Parallel Estimation of Distribution Algorithms* by Alexander Mendiburu, José Miguel-Alonso and José A. Lozano.

Estimation of distribution algorithms (EDAs) are a class of evolutionary algorithms that learn a probabilistic model from the individuals selected in a given generation. Instead of the usual genetic operators of crossover and mutation, EDAs create new individuals by sampling from the learned probabilistic model.

The chapter by Mendiburu, Miguel-Alonso and Lozano begins with an introduction to EDAs and a summary of parallel computing technology. Next, the authors identify two basic approaches for parallelizing EDAs: direct and island-based. The rest of the chapter is devoted to describe several algorithms that follow each of the approaches.

Mendiburu et al. identify the parts that consume the most time as learning the probabilistic model and evaluating the individuals. Parallelizing the evaluation of individuals is simple using a manager-worker scheme, but learning the probabilistic model can be more complicated. The authors describe increasingly complex probabilistic models: model without dependencies, and models that consider all possible dependencies. The first class of models are easy to parallelize by splitting the work required across of variables opens the possibility for directly distributing work to several computing nodes. For the more sophisticated models, the authors offer a review of algorithms that learn the model structure and its parameters.

On the island-based approaches, the authors recognize that EDAs can use the same paradigm that other EAs use for exchanging information through migration of individuals, but can also exchange the probabilistic models learned in each island. There is some empirical work that suggests that exchanging and combining models instead of individuals offers some advantages, but in general there are many opportunities to expand the research in this area, and Mendiburu et al. identify some future avenues for research.

Chapter 7. *Parallel Multi-Objective Optimization using Self-Organized Heterogeneous Resources* by Sanaz Mostaghim.

In this chapter, Mostaghim shows a multi-objective optimization algorithm on heterogeneous computing resources. The focus is on a multiobjective particle swarm algorithm. In the approach presented in this chapter, the user is not involved in task partitioning, scheduling, or synchronization. Instead the heterogeneous computing resources are presented as a unified resource.

Mostaghim studies how to automatically partition the optimization task between the available processors. In the partitioning every computing resource is responsible for one part of the optimal front. Tasks are also scheduled automatically and the algorithm takes into account the heterogeneity of the computing resources. Synchronization happens using the global archive and different nodes look for areas of the optimal front that are less populated and start optimizing on those areas.

Mostaghim demonstrates his approach with experiments on an environment containing 100 computing nodes of five different types. The experiments measure the solution quality over a range of waiting times (how long before fast processors wait for slow ones) and artificial failures on the system intended to simulate realistic collections of heterogeneous computing resources.

Chapter 8. *The Role of Explicit Niching and Communication Messages in Distributed Evolutionary Multi-Objective Optimization* by Lam T. Bui, Daryl Essam and Hussein A. Abbass.

In this chapter Bui, Essam, and Abbass consider an Evolutionary Multi-objective Optimization framework in which each sub-area of the search space is associated with a separate population and is used to build a local model. These models are allowed to interact using rules inspired from Particle Swarm Optimization.

This chapter investigates several aspects of the communication between populations. First, it examines the effect of the contents of the messages. In most parallel and distributed EAs, it is common to communicate subsets of individuals, but the communication cost can be reduced by using summary information of the system's progress instead of the solutions themselves. The chapter also investigates the effect of the frequency of the communications and considers different communication architectures.

The effect of the content of the messages has not been studied in detail before. Bui et al. consider three choices for the contents of messages: (1) local non-dominated solutions that are used to build a global archive to guide the system, (2) local directions of improvement that are used to build a global direction of the system, and (3) the direction of improvement and the size of the non-dominated sets, which is used to weight the contribution of each local direction to the global direction. For all three kinds of messages, the authors examine the communication costs in master-slave, island, and a hybrid architecture of several master-slave processes connected.

The effectiveness and efficiency of the different combinations of messages and architectures is tested using several benchmark functions. The authors also propose several approximate equations that can be helpful in understanding the scalability of the different communication alternatives.

Chapter 9. *Adaptive Scheduling Algorithms for the Dynamic Distribution and Parallel Execution of Spatial Agent-Based Models* by Matthias Scheutz and Jack Harris.

Spatial agent-based models lend themselves to efficient parallel implementations. These models explicitly define an environment, which is usually a metric space, and every agent is situated in a particular location in the environment at any given time. Each agent has a state that is generally updated on every cycle of the simulation using information about the current state and perhaps the state of other agents. Every agent also has an interaction range that, given the agent's location, determines at any given time the set of other agents in the environment with which it can interact.

One way to parallelize these models is to distribute simulations across multiple computers. Scheutz and Harris propose a novel update scheduling algorithm for simulations of spatial agent-based models. The algorithms take advantage of the inherent parallelism in agent-based models and the interaction ranges of agents. The authors propose four heuristic methods for updating the states of the agents and show that three of them can improve performance over the standard update strategy that is based on collecting state information from all the other agents in each cycle. In essence, the heuristics try to determine in advance which agents need each others' state information and distribute the agents based on these dependencies. The proposed heuristics permit for agents to be at different cycles within one simulation, as long as the cycle differences do not lead to inconsistent update sequences. The idea is that the simulation can update the state of some agents while others wait for information from other computing nodes necessary for their update.

Scheutz and Harris experiment with an agent-based simulation taken from a biological domain and achieve from than 50% shorter simulation run times.

Chapter 10. *On the Use of Distributed Genetic Algorithms for the Tuning of Fuzzy Rule Based-Systems* by Ignacio Robles, Rafael Alcalá, José M. Benítez and Francisco Herrera.

In their chapter, Robles, Alcalá, Benítez and Herrera present a study using distributed genetic algorithms to tune fuzzy rule-based systems. In particular, the authors consider linguistic fuzzy rule-based systems with two components: a rule base and a database. The rule base that contains rules of the general form IF antecedent(s) THEN consequent(s). The database contains the linguistic term sets used in linguistic rules as well as the membership functions that define the semantics of linguistic labels. For example, for a linguistic variable such as temperature, there can be three fuzzy sets corresponding to low, medium, and high temperatures. The membership functions of the three fuzzy sets are defined by certain parameters that need to be tuned to specific applications.

For tuning the membership functions, Robles et al. use an algorithm they call gradual distributed Real-Coded Genetic Algorithm. The algorithm uses eight subpopulations with different genetic operators. The basic idea is that some of the subpopulations will do aggressive exploration, and communicate individuals to subpopulations dedicated to exploitation.

The authors examine the performance of their approach using four benchmark problems and compare it to the performance of a sequential algorithm. The results suggest that complex problems benefit more from the distributed approach.

Chapter 11. *Parallel and distributed optimization of dynamic data structures for multimedia embedded systems* by José L. Risco Martín and David Atienza and J. Ignacio Hidalgo and Juan Lanchares.

One of the biggest challenges to implement multimedia applications on embedded devices is the limited memory available. Applications that run on desktops usually do not pay attention to power consumption, memory access patterns, or memory usage, but these are important factors in embedded systems. Therefore, to implement applications, designers must find the best set of dynamic data types that minimizes cost and obeys the constraints of the target device. Risco Martín, Atienza, Hidalgo and Lanchares present a method to optimize dynamic data structures for multimedia embedded systems based on a parallel multi-objective EA.

They explore different classical MOEAs and propose an algorithm that combines NSGA-II and SPEA2, two well-known MOEAs. Their algorithm follows an island approach, where each island executes a different MOEA and individuals migrate between islands every 100 generations.

Risco Martín et al. perform experiments in a real-life dynamic embedded application and show that NSGA-II and SPEA2 reach huge speed-ups (up to 469X faster) with respect to traditional heuristics and the parallel algorithms achieve significant speedups with respect to the sequential versions in a multi-core architecture. Moreover, using multiple metrics, the authors show that the quality of the solutions is improved by the combination of NSGAII and SPEA2 in a parallel implementation. In terms of scalability, the authors obtained empirical evidence that shows that on increasing the size of the population, the performance of the parallel MOEA improves as more workstations are used.

Chapter 12. *A Grid-based Hybrid Hierarchical Genetic Algorithm for Protein Structure Prediction* by Alexandru-Adrian Tantar, Nouredine Melab, El-Ghazali Talbi.

In this chapter Tantar, Melab and Talbi address the problem of protein structure prediction with a combination of an EA and Adaptive Simulated Annealing. This is a very difficult problem that consists of determining the conformation of a given protein given its aminoacid sequence.

To address this problem, the authors present a hierarchical hybrid parallel genetic algorithm. The hierarchical parallelization consists of using an island model GA at the highest level, then parallelizing the fitness evaluations in each island, and furthermore each evaluation is executed on several processors. The hybridization consists of using conjugate gradient optimization as an additional search operator and using a synchronous adaptive simulated annealing on a subset of the individuals after a fixed number of generations.

Tantar et al. present an implementation on up to a thousand cores distributed in different grids across several academic institutions. The authors describe increasingly sophisticated approaches to their hybridization. Their experiments with their initial parallel hybrid approach did not obtain very good results, so they experimented with alternative representations and free energy evaluation functions. The representations and the evaluation functions required additional tuning that was performed with a meta-evolutionary algorithm. Each individual of the meta-EA was composed of multiple islands, each with parallel fitness evaluations. With the meta-EA implementation the authors obtained very good solutions for two of the three molecules that were investigated.

Chapter 13. *Laser Dynamics Modelling and Simulation: An application of Dynamic Load Balancing of Parallel Cellular Automata* by J.L. Guisado, F. Jiménez-Morales, J.M. Guerra, F. Fernández de Vega, K.A. Iskra, P.M.A. Sloot, and Daniel Lombraña González.

This chapter is an application of cellular automata to the modeling of laser dynamics. The authors create a set of transition rules for the cellular automata that can reproduce different aspects of laser systems. Their experimental results show that there is qualitative agreement between the simulations and the behavior expected from modeling the laser with differential equations. To pursue more realistic simulations, larger lattices are needed and make it necessary to use parallel implementations. The authors first study the performance of a master-worker version of their algorithm on a small computer cluster and find good speedups and scalability results. Next, the authors study the question of the efficiency of the algorithms on non-dedicated heterogeneous clusters of computers using a dynamic load balancing approach. The authors added artificial tasks to the cluster and observed that the dynamic load balancing made a large difference in execution time, reducing it by a factor of almost five compared to not using load balancing.

When Huge Is Routine: Scaling Genetic Algorithms and Estimation of Distribution Algorithms via Data-Intensive Computing

Xavier Llorà, Abhishek Verma, Roy H. Campbell, and David E. Goldberg

Abstract. Data-intensive computing has emerged as a key player for processing large volumes of data exploiting massive parallelism. Data-intensive computing frameworks have shown that terabytes and petabytes of data can be routinely processed. However, there has been little effort to explore how data-intensive computing can help scale evolutionary computation. In this book chapter we explore how evolutionary computation algorithms can be modeled using two different data-intensive frameworks—Yahoo!'s Hadoop and NCSA's Meandre. We present a detailed step-by-step description of how three different evolutionary computation algorithms, having different execution profiles, can be translated into the data-intensive computing paradigms. Results show that (1) Hadoop is an excellent choice to push evolutionary computation boundaries on very large problems, and (2) that transparent Meandre linear speedups are possible without changing the underlying data-intensive flow thanks to its inherent parallel processing.

1 Introduction

Data-intensive computing branding is relatively recent, however data flow started to get traction back in the mid 90's with the appearance of frameworks such as IBM

Xavier Llorà
National Center for Supercomputing Applications,
University of Illinois at Urbana-Champaign, 1205 W. Clark Street, Urbana, IL 61801
e-mail: xllora@illinois.edu

Abhishek Verma · Roy H. Campbell
Department of Computer Science, University of Illinois at Urbana-Champaign,
201 N Goodwin Ave, Urbana, IL 61801
e-mail: {verma7,rhc}@illinois.edu

David E. Goldberg
Department of Industrial and Enterprise Systems Engineering,
University of Illinois at Urbana-Champaign, 104 S. Mathews Ave, Urbana, IL 61801
e-mail: deg@illinois.edu

promoted FBP[33] or NCSA's D2K [45], and later simplified and popularized by Google's MapReduce model [11] and Yahoo!'s open source Hadoop project[1]. Recent advances on data-intensive computing have lead to frameworks that are now able to exploit massive parallelism to efficiently process petabytes of data. These frameworks, due to their data-flow nature, provide specialized programming environments tailored for developing flow applications that, up to a certain degree, transparently benefit from the available parallelism.

The current data deluge is happening across different domains and is forcing a rethinking of how large volumes of data are processed. Most of these data-intensive computing frameworks share a common underlying characteristic: data-flow oriented processing. Availability of data drives, not only the execution, but also the parallel nature of such processing. The growth of the internet and its easy communication medium has pushed researchers from all disciplines to deal with volumes of information where the only viable path is to utilize data-intensive frameworks [43, 6, 14, 32]. Although large bodies of research on parallelizing evolutionary computation algorithms are available [8, 1], there has been little work done in exploring the usage of data-intensive computing [23, 28].

The inherent parallel nature of evolutionary algorithms makes them optimal candidates for parallelization [8]. Moreover, as we will layout in this paper, evolutionary algorithms and their inherent need to deal with large volumes of data—regardless if it takes the form of populations of individuals or samples out of a probabilistic distribution—can greatly benefit from a data-intensive computing modeling. In this book chapter we will explore the usage of two frameworks: Yahoo!'s Hadoop model and its MapReduce implementation, and NCSA's semantic-driven data-intensive computing framework – Meandre[2] [30]. Hadoop provides a simple scalable programming model based on the implementation of two basic functions: the *map* and the *reduce* functions. The *map* function provides uniform and parallel process to large volumes of data in forms of chunks, whereas the *reduce* function aggregates the results produced by mappers. On the other hand, Meandre allows explicit descriptions of complex, and possibly iterating, data flows via a directed multigraph of components describing the data flow processing. Two main benefits of such a modeling are: (1) favoring encapsulation, reutilization, and sharing via Lego-like component modeling, and (2) massive parallel data-driven execution. The first benefit targets improving software engineering best practices and a detailed discussion is beyond the scope of this paper and can be find elsewhere [30].

To illustrate the benefits for the evolutionary computation community of adopting such approaches we selected three representative algorithms and developed their equivalent data-intensive computing equivalents. It is important to note here that we paid special attention to guarantee that the underlaying mechanics were not altered and the properties of these algorithms maintained. The three algorithms transformed were: a simple selectorecombinative genetic algorithm [16, 17], the compact genetic algorithm [21], and the extended compact genetic algorithm [22]. We will show how

[1] `http://hadoop.apache.org/`

[2] Catalan spelling of the word *meander*. `http://seasr.org/meandre`

a simple selectorecombinative genetic algorithm [16, 17] can be modeled using the data-intensive computing via Hadoop's MapReduce approach and Meandre data-intensive flow modeling. We will review (1) some of the basic steps of the transformation process required to achieve its data-intensive computing counterparts, (2) how to design components that can maximally benefit from a data-driven execution, and (3) analyze the results obtained. The second example, the compact genetic algorithm [21], we focus on how Hadoop's MapReduce modeling can help scale being a clear competitor of traditional high performance computing version [41]. The third example addresses the parallelization of the model building of estimation of distribution algorithms. We will show how Meandre's data-driven implementation of the extended compact classifier system (eCGA) [22] produces, *de facto*, a parallelized implementation of the costly model building stage. Experiments show that speedups linear to the number of cores or processors are possible without any further modification.

It is important to note here, that each of these algorithms has different profiles. For instance, the simple selectorecombinative genetic algorithm requires dealing with large populations as you tackle large problems, but the operators are straight forward. The compact genetic algorithm instead is memory efficient, but requires the proper updating of a simple probability distributions. Finally the extended compact genetic algorithms requires to deal with large populations as you scale your problem size, and also requires an elaborated model building process to induce the probability distribution required. In this book chapter, we will focus on the massive parallel data-driven execution that allows users to automatically benefit from the advances of the current multicore era — which has opened the door to petascale computing — without having to modify the underlying algorithm.

The rest of this chapter is organized as follows. Section 2 presents a quick introductory overview of the two data-intensive frameworks we will use through the chapter, Hadoop and Meandre. Then, Section 3 introduces the three evolutionary computation algorithms that we will use in our experimentation with the two introduced frameworks, a simple selectorecombinative genetic algorithm , the compact genetic algorithm, and the extended compact genetic algorithm. These algorithms are transformed and implemented using data-intensive computing techniques, and the proposed implementations are discussed on Section 4. Section 5 presents the results achieved and using the data-intensive implementations showing that scalability is only bounded by the available resources, and linear speedups are easily achievable. Finally we review some related work in section 6 and present some conclusions and possible further work on section 7.

2 Data-Intensive Computing

This section presents a quick overview of the two data-intensive frameworks we will use throughout the rest of this book chapter. The first one is Hadoop[3] — Yahoo!'s open source MapReduce framework. Modeled after Google's MapReduce

[3] http://hadoop.apache.org

paper [11], Hadoop builds on the *map* and *reduce* primitives present in functional languages. Hadoop relies on these two abstractions to enable the easily development of large-scale distributed applications as long as your application can be modeled around these two phases. The second framework is Meandre [30]—NCSA's data-intensive computing infrastructure for science, engineering, and humanities. Meandre provides a more flexible programming model that allows to create complex data flows, which could be regarded as complex and possible iterating MapReduce stages. Meandre can also benefit of some Hadoop tools, such as Hadoop's distributed file system.

2.1 MapReduce and the Hadoop Model

Inspired by the *map* and *reduce* primitives present in functional languages, Google popularized the MapReduce[11] abstraction that enables users to easily develop large-scale distributed applications. The associated implementation parallelizes large computations easily as each map function invocation is independent and uses re-execution as the primary mechanism of fault tolerance.

In this model, the computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce framework then groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. The intermediate values are supplied to the user's reduce function via an iterator. This allows the model to handle lists of values that are too large to fit in main memory.

Conceptually, the map and reduce functions supplied by the user have the following types:

$$\begin{aligned} map(k_1, v_1) &\rightarrow list(k_2, v_2) \\ reduce(k_2, list(v_2)) &\rightarrow list(v_3) \end{aligned}$$

i.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function, which is $hash(key)\%R$ according to the default Hadoop configuration. The number of partitions (R) and the partitioning function are specified by the user. The overall

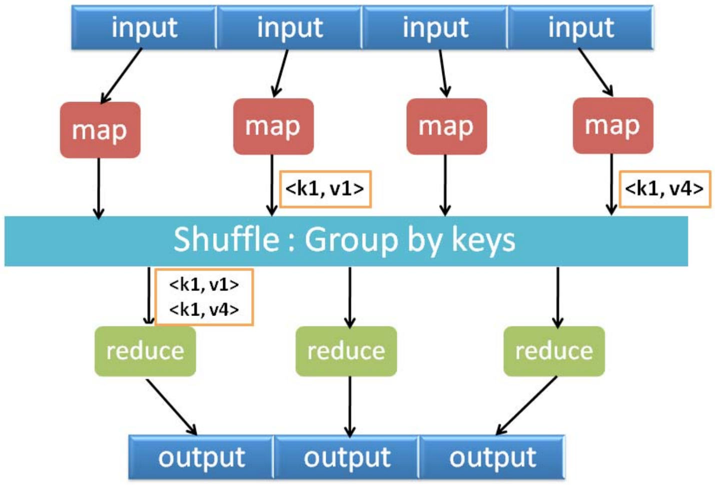


Fig. 1. MapReduce data flow overview

execution is thus orchestrated in two steps: first all mappers are executed in parallel, then the reducers process the generated key value pairs by the reducers. A detailed explanation of this framework is beyond the scope of this article and can be found elsewhere [11]. We will also use the Yahoo!'s open source MapReduce framework through this article.

2.2 Data-Intensive Flow Computing with Meandre

Meandre [30] is a semantic-enabled web-driven, dataflow execution environment. It provides the machinery for assembling and executing data flows. Flows are software applications composed by components that process data. Each flow represents as a directed multigraph of executable components—nodes—linked through their input and output ports. Based on the inputs, properties, and its internal state, an executable component may produce output data. Meandre also provides component and flow publishing capabilities enabling users to assemble a repository of components by reusing and sharing. Users can discover by querying and reuse components and flows previously published by other researchers. It is important to mention here, that component and flow abstract can act as self-contained elements—other approaches like Chimera still rely on external information [14]. Meandre builds on three main concepts: (1) dataflow-driven execution, (2) semantic-web metadata manipulation, and (3) metadata publishing. A detailed description of the Meandre data-intensive computing architecture is beyond the scope of this paper and can be found elsewhere [30].

2.2.1 Dataflow Execution Engines

Conventional programs perform their computational tasks by executing a sequence of instructions. One after another, each code instruction is fetched and executed. All data manipulation is performed by these basic units of execution. In a broad sense, this approach can be termed "code-driven execution." Any computation task is regarded as a sequence of code instructions that ultimately manipulates data. However, data-driven execution (or dataflow execution) revolves around the idea of applying transformational operations to a flow or stream of data. In a data-driven model, data availability determines the sequence of code instructions to execute.

An analogy of the dataflow execution model is the black box operand approach. That is, any operand (operator) may have zero or more data inputs. It may also produce zero or more data through its data outputs. The operand behavior may be controlled by properties (behavior controls). Each operand performs its operations based on the availability of its inputs. For instance, an operand may require that data is available in all its inputs to perform its operations. Others may only need some, or none. A simple example of a black box operand could be the arithmetic '+' operand. This operand can be modeled as follows:

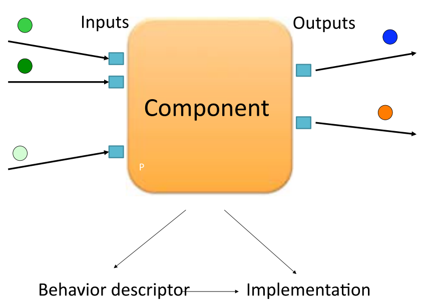
1. It requires two inputs.
2. When two inputs are available, it performs the addition.
3. It then pushes the result to an output.

Such a simple operand may have two possible implementations. The first one defines a executable component (Meandre terminology for a black box operator) with two inputs. When data is present on both inputs, then the operator is executed—*fired*. The operator produces one piece of data to output, which may become the input of another operator. Another possible implementation is to create a component with a single input that adds together two consecutive data pieces received. The component requires an internal variable (or state) which stores the first data piece of a pair. When the second data piece arrives, it would be added to the first and an output is produced. The internal variable would then be cleared and the component will treat the next data piece received as the first of a new pair. As we will see later in this paper, both implementations have merit, but in certain conditions we will choose one over the other based on clarity and efficiency requirements.

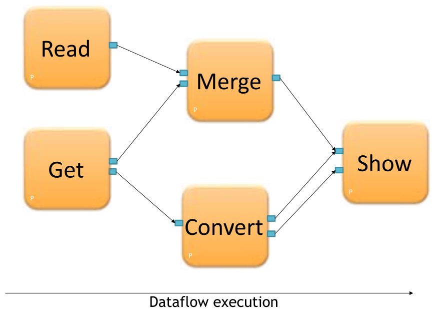
Meandre uses the following terminology:

1. *Executable component*: A basic unit of processing.
2. *Input port*: Input data required by a component.
3. *Firing policy*: The policy of component execution (e.g. when all/any input ports contain data).
4. *Output port*: Outputs data produced by component execution.
5. *Properties*: Component read-only variables used to modify component behavior.
6. *Internal state*: The collection of data structures designed to manage data between component firings.

Figure 2 presents a schema of the component and flow anatomy. Components with input and output ports can be connected to describe a complex task, commonly



(a) A component is described by several input and output ports where data flows through. Also, each component have a set of properties which govern its behavior in the presence of data.



(b) A flow is a directed graph where multiple components are connected together via input/output ports. A flow represents a complex task to solve.

Fig. 2. A data-intensive flow is characterized by the components it uses (basic process units) and their interconnection (a direct multigraph). Grouping several components together describes a complex task. It also emphasize rapid development by component reutilization.

referred as flow. Dataflow execution engines provide a scheduler that determines the firing (execution) sequence of components[4].

2.2.2 Components

Meandre components serve as the basic *building block* of any computational task. There are two kinds of Meandre components: (1) *executable components* and (2) *flow components*. Regardless of type, all Meandre components are described using metadata expressed in RDF. Executable components also require an executable implementation form that can be understood by the Meandre execution engine[5]. Meandre's metadata relies on three ontologies: (1) the RDF ontology [5, 7] serves as a base for defining Meandre components; (2) the Dublin Core elements ontology [44] provides basic publishing and descriptive capabilities in the description of Meandre components; and (3) the Meandre ontology describes a set of relationships that model valid components, as understood by the Meandre architecture—refer to [30] for a more detailed description.

2.2.3 Programming Paradigm

The programming paradigm creates complex tasks by linking together a bunch of specialized components—see Figure 2. Meandre's publishing mechanism allows

[4] Meandre uses a *decentralized scheduling policy* designed to maximize the use of multicore architectures. Meandre also allows works with processes that require directed cyclic graphs—extending beyond the traditional MapReduce directed acyclic graphs.

[5] Java, Python, and Lisp are the current languages supported by Meandre to implement a component.

components developed by third parties to be assembled in a new flow. There are two ways to develop flows on Meandre: (1) using visual programming tools, or (2) using Meandre's ZigZag scripting language—see [30]. For simplicity purposes, throughout the rest of this paper flows will be presented as ZigZag scripts.

3 Three Diverse Genetic Algorithms Models

Evolutionary computing encompass a large diversity of algorithms and implementations. In order to illustrate the usefulness of data-intensive computing, we will focus on three widely used models: selectorecombinative genetic algorithms [16, 17], the compact genetic algorithm [21], and the extended compact genetic algorithm [22]. As we will describe in the rest of this section, each of these algorithms posses different profiles. Ranging from purely population-based to model-based algorithms, to create their data-intensive computing counterparts—as we will show in the next section—will require to pay close attention to their basic needs.

3.1 A Simple Selectorecombinative Genetic Algorithm

Selectorecombinative genetic algorithms [16, 17] mainly rely on the use of selection and recombination. We chose to start with them because they present a minimal set of operators that will help us illustrate the creation of a data-intensive flow counterpart. As we will see, the addition of mutation operators will be trivial after the setting up the proper data-intensive flow. The rest of this section will present a quick description of the algorithm we transformed and implemented it using Meandre, a discussion of some of the elements that need to be taken into account, and finally review the execution profile of the final implementation.

The basic algorithm we will target to implement as a data-intensive flow can be summarized as follows:

1. Initialize the population with random individuals.
2. Evaluate the fitness value of the individuals.
3. Select good solutions by using s-wise tournament selection without replacement [19].
4. Create new individuals by recombining the selected population using uniform crossover[6] [42].
5. Evaluate the fitness value of all offspring.
6. Repeat steps 3–5 until some convergence criteria are met.

3.2 The Compact Genetic Algorithm

The compact genetic algorithm [21], is one of the simplest estimation distribution algorithms (EDAs) [36, 24]. Similar to other EDAs, cGA replaces traditional variation

[6] For this particular exercise we have assumed a crossover probability p_χ=1.0.

operators of genetic algorithms by building a probabilistic model of promising solutions and sampling the model to generate new candidate solutions. The probabilistic model used to represent the population is a vector of probabilities, and therefore implicitly assumes each gene (or variable) to be independent of the other. Specifically, each element in the vector represents the proportion of ones (and consequently zeros) in each gene position. The probability vectors are used to guide further search by generating new candidate solutions variable by variable according to the frequency values.

The compact genetic algorithm consists of the following steps:

1. *Initialization:* As in simple GAs, where the population is usually initialized with random individuals, in cGA we start with a probability vector where the probabilities are initially set to 0.5. However, other initialization procedures can also be used in a straightforward manner.
2. *Model sampling:* We generate two candidate solutions by sampling the probability vector. The model sampling procedure is equivalent to uniform crossover in simple GAs.
3. *Evaluation:* The fitness or the quality-measure of the individuals are computed.
4. *Selection:* Like traditional genetic algorithms, cGA is a selectionist scheme, because only the better individual is permitted to influence the subsequent generation of candidate solutions. The key idea is that a "survival-of-the-fittest" mechanism is used to *bias* the generation of new individuals. We usually use tournament selection [19] in cGA.
5. *Probabilistic model update:* After selection, the proportion of winning alleles is increased by $1/n$. Note that only the probabilities of those genes that are different between the two competitors are updated. That is,

$$p_{x_i}^{t+1} = \begin{cases} p_{x_i}^t + 1/n & \text{If } x_{w,i} \neq x_{c,i} \text{ and } x_{w,i} = 1, \\ p_{x_i}^t - 1/n & \text{If } x_{w,i} \neq x_{c,i} \text{ and } x_{w,i} = 0, \\ p_{x_i}^t & \text{Otherwise.} \end{cases} \quad (1)$$

   Where, $\mathbf{x}_{w,i}$ is the i^{th} gene of the winning chromosome, $\mathbf{x}_{c,i}$ is the i^{th} gene of the competing chromosome, and $p_{x_i}^t$ is the i^{th} element of the probability vector—representing the proportion of i^{th} gene being one—at generation t. This updating procedure of cGA is equivalent to the behavior of a GA with a population size of n and steady-state binary tournament selection.
6. Repeat steps 2–5 until one or more termination criteria are met.

The probabilistic model of cGA is similar to those used in population-based incremental learning (PBIL) [3, 4] and the univariate marginal distribution algorithm (UMDA) [35, 34]. However, unlike PBIL and UMDA, cGA can simulate a genetic algorithm with a given population size. That is, unlike the PBIL and UMDA, cGA modifies the probability vector so that there is direct correspondence between the population that is represented by the probability vector and the probability vector itself. Instead of shifting the vector components proportionally to the distance from either 0 or 1, each component of the vector is updated by shifting its value by the contribution of a single individual to the total frequency assuming a particular population size.

Additionally, cGA significantly reduces the memory requirements when compared with simple genetic algorithms and PBIL. While the simple GA needs to store n bits, cGA only needs to keep the proportion of ones, a finite set of n numbers that can be stored in $\log_2 n$ for each of the ℓ gene positions. With PBIL's update rule, an element of the probability vector can have any arbitrary precision, and the number of values that can be stored in an element of the vector is not finite.

Elsewhere, it has been shown that cGA is operationally equivalent to the order-one behavior of simple genetic algorithm with steady state selection and uniform crossover [21]. Therefore, the theory of simple genetic algorithms can be directly used in order to estimate the parameters and behavior of the cGA. For determining the parameter n that is used in the update rule, we can use an approximate form of the gambler's ruin population-sizing[7] model proposed by Harik et al. [20]:

$$n = -log\alpha \cdot \frac{\sigma_{BB}}{d} \cdot 2^{k-1}\sqrt{\pi \cdot m}, \qquad (2)$$

where k is the BB size, m is the number of building blocks (BBs)—note that the problem size $\ell = k \cdot m$,—d is the size signal between the competing BBs, and σ_{BB} is the fitness variance of a building block, and α is the failure probability.

3.3 The Extended Compact Genetic Algorithm

The extended compact genetic algorithm (eCGA) [22], is based on a key idea that the choice of a good probability distribution is equivalent to linkage learning. The measure of a good distribution is quantified based on minimum description length (MDL) models. The key concept behind MDL models is that given all things are equal, simpler distributions are better than the complex ones. The MDL restriction penalizes both inaccurate and complex models, thereby leading to an optimal probability distribution. The probability distribution used in eCGA is a class of probability models known as marginal product models (MPMs). MPMs are formed as a product of marginal distributions on a partition of the genes. MPMs also facilitate a direct linkage map with each partition separating tightly linked genes.

The eCGA, later extended to deal with n-ary alphabets in χ-eCGA [10], can be algorithmically outlined as follows:

1. Initialize the population with random individuals.
2. Evaluate the fitness value of the individuals.
3. Select good solutions by using s-wise tournament selection without replacement [19].
4. Build the probabilistic model: In χ-eCGA, both the structure of the model as well as the parameters of the models are searched. A greedy search is used to search for the model of the selected individuals in the population.

[7] The experiments conducted in this paper used $n = 3\ell$.

5. Create new individuals by sampling the probabilistic model.
6. Evaluate the fitness value of all offspring.
7. Repeat steps 3–6 until some convergence criteria are met.

Two things need further explanation: (1) the identification of MPM using MDL, and (2) the creation of a new population based on MPM.

The identification of MPM in every generation is formulated as a constrained optimization problem,

$$\text{Minimize} \quad C_m + C_p \tag{3}$$

$$\text{Subject to}$$

$$\chi^{k_i} \le n \quad \forall i \in [1,m] \tag{4}$$

where χ is the alphabet cardinality—$\chi = 2$ for the binary strings—C_m is the model complexity which represents the cost of a complex model and is given by

$$C_m = \log_\chi(n+1) \sum_{i=1}^{m} \left(\chi^{k_i} - 1\right) \tag{5}$$

and C_p is the compressed population complexity which represents the cost of using a simple model as against a complex one and is evaluated as

$$C_p = \sum_{i=1}^{m} \sum_{j=1}^{\chi^{k_i}} N_{ij} \log_\chi \left(\frac{n}{N_{ij}}\right) \tag{6}$$

where m in the equations represent the number of BBs, k_i is the length of BB $i \in [1,m]$, and N_{ij} is the number of chromosomes in the current population possessing bit-sequence $j \in [1,\chi^{k_i}]$[8] for BB i. The constraint (Equation 4) arises due to finite population size.

The greedy search heuristic used in χ-eCGA starts with a simplest model assuming all the variables to be independent and sequentially merges subsets until the MDL metric no longer improves. Once the model is built and the marginal probabilities are computed, a new population is generated based on the optimal MPM as follows, population of size $n(1-p_c)$ where p_c is the crossover probability, is filled by the best individuals in the current population. The rest $n \cdot p_c$ individuals are generated by randomly choosing subsets from the current individuals according to the probabilities of the subsets as calculated in the model.

One of the critical parameters that determines the success of eCGA is the population size. Analytical models have been developed for predicting the population-sizing and the scalability of eCGA [40]. The models predict that the population size required to solve a problem with m building blocks of size k with a failure rate of $\alpha = 1/m$ is given by

[8] Note that a BB of length k has χ^k possible sequences where the first sequence denotes be $00\cdots0$ and the last sequence $(\chi-1)(\chi-1)\cdots(\chi-1)$.

$$n \propto \chi^k \left(\frac{\sigma_{BB}^2}{d^2}\right) m \log m, \tag{7}$$

where n is the population size, χ is the alphabet cardinality (here, $\chi = 3$), k is the building block size, $\frac{\sigma_{BB}^2}{d^2}$ is the noise-to-signal ratio [18], and m is the number of building blocks. For the experiments presented in this paper we used $k = |a| + 1$ (where $|a|$ is the number of address inputs), $\frac{\sigma_{BB}^2}{d^2}$=1.5, and $m = \frac{\ell}{|I|}$ (where ℓ is the rule size).

4 Data-Intensive Computing in Action

The previous section described the three algorithms we will target to create their data-intensive computing counterparts. This section we will take a stab at designing efficient and scalable version of these algorithms to show the benefits of banking on either MapReduce or Meandre approaches.

4.1 A Simple Selectorecombinative Genetic Algorithm

4.1.1 MapReducing SGAs

In this section, we start with a simple model of Genetic algorithms and then transform and implement it using MapReduce. We encapsulate each iteration of the GA as a seperate MapReduce job. The client accepts the commandline parameters, creates the population and submits the MapReduce job.

Map: Evaluation of the fitness function for the population matches the MAP function, which has to be computed independent of other instances. As shown in the algorithm in Listing 1, the MAP evaluates the fitness of the given individual. Also, it keeps track of the the best individual and finally, writes it to a global file in the Distributed File System (HDFS)[9]. The client reads these values from all the mappers at the end of the MapReduce and determines whether to start the next iteration.

Partitioner: If the selection operation in a GA (Step 3) is performed locally on each node, it reduces the selection pressure [38] and can lead to increase in the time taken to converge. Hence, decentralized and distributed selection algorithms [9] are preferred. The only point at which there is a global communication is in the shuffle between the Map and Reduce. At the end of the Map phase, the MapReduce framework shuffles the key/value pairs to the reducers using the partitioner. The partitioner splits the intermediate key/value pairs among the reducers. The function GETPARTITION() returns the reducer to which the given $(key, value)$ should be sent to. In the default implementation, it uses HASH(key) %

[9] This cleanup functionality can be implemented by overriding the *close*() function, but it's overlaps with *reduce*() function and hence sometimes throws *FileNotFoundException*.

Listing 1. Map phase of each iteration of the Genetic Algorithm

```
procedure Initialization:
begin
  max := −1
end

procedure Map(key, value):
begin
  individual := Individual_representation(key)
  fitness := CalculateFitness(individual)
  Emit (individual, fitness)
  {Keep track of the current best}
  if fitness>max then
    max := fitness
    maxInd := individual
  {Finished all local maps}
  if processed_all_individuals then
    Write best individual to global file in DFS
end
```

numReducers so that all the values corresponding to a given *key* end up at the same reducer which can then apply the REDUCE function. However, this does not suit the needs of Genetic algorithms because of two reasons: Firstly, the HASH function partitions the namespace of the individuals N into r distinct classes : $\{N_0, N_1, \ldots, N_{r-1}\}$ where $N_i = \{n : \textsc{Hash}(n) = i\}$. The individuals within each partition are isolated from all other partitions. Thus, the HASHPARTITIONER introduces an artificial spatial constraint based on the lower order bits. Because of this, the convergence of the genetic algorithm may take more iterations or it may never converge at all.

Secondly, as the genetic algorithm progresses, the same (close to optimal) individual begins to dominate the population. All copies of this individual will be sent to one single reducers which will get overloaded. Thus, the distribution progressively becomes more skewed, deviating from the uniform distribution (that would have maximized the usage of parallel processing). Finally, when the GA converges, all the individuals will be processed by that single reducer. Thus, the parallelism decreases as the GA converges and hence, it will take more iterations.

For these reasons, we override the default partitioner by providing our own partitioner, which shuffles individuals randomly across the different reducers as shown in Listing 2.

Reducer: We implement Tournament selection without replacement[19]. A tournament is conducted among *tSize* randomly chosen individuals and the winner is selected. This process is repeated *population* number of times. Since randomly selecting individuals is equivalent to randomly shuffling all individuals and then processing them sequentially, our reduce function goes through the individuals

Listing 2. Random partitioner for the Genetic Algorithm

```
int getPartition(key, value, numReducers):
    return RandomInt(0, numReducers − 1)
```

sequentially. Initially the individuals are buffered for the last rounds, and when the tournament window is full, SELECTIONANDCROSSOVER is carried out as shown in the Listing 3. When the crossover window is full, we use the Uniform Crossover operator. For our implementation, we set the *tSize* to 5 and the *cSize* to 2.

Optimizations: After initial experimentation, we noticed that for larger problem sizes, the serial initialization of the population takes a long time. According to Amdahl's law, the speedup is bounded because of this serial component. Hence, we create the initial population in a separate MapReduce phase, in which the MAP generates random individuals and the REDUCE is the Identity Reducer. [10] We seed the pseudo-random number generator for each mapper with $mapperId \cdot currentTime$. The bits of the variables in the individual are compactly represented in an array of **long long int**s and we use efficient bit operations for crossover and fitness calculations. Due to the inability of expressing loops in the MapReduce model, each iteration consisting of a Map and Reduce, has to executed till the convergence criteria is satisfied.

4.1.2 SGAs as Data-Intensive Flows

The first step in designing a data-intensive flow implementation of the algorithm presented in the previous section is to identify what data will be processed. This decision is similar to the *partition* step of the methodology proposed by Foster [13], to design general purpose parallel programs, where data is maximally partitioned to maximize parallelization. The two possible options here are to deal with populations or individuals. E2K [27]—a data-flow extension for D2K [45]—chose to use populations. Such a decision only allows parallelizing the concurrent evolution of distinct populations. In this paper we will choose the second option. Our data-flow implementation is going to be built around processing individuals. In other words, a population will be a stream of individuals—a *stream initiator* and a *stream terminator* will enclose each stream defining a population. Making the decision of processing streams of individuals will allow creating components that perform the genetic manipulation as the stream goes by. This approach may be regarded as an analogy of pipeline segmentation on central processing units.

Inspecting the algorithm presented in section 3.1, we need to create components that implement each of the operation required: *initialization*, *evaluation*, *selection*, and *recombination*. *Initialization* and *evaluation* are straight forward; the first one

[10] Setting the number of reducers to 0 in Hadoop removes the extra overhead of shuffling and identity reduction.

Listing 3. Reduce phase of each iteration of the Genetic Algorithm

```
procedure Initialization:
begin
    processed := 0
    Allocate tournamentArray [1 ... 2*tSize]
    Allocate crossoverArray [cSize]
end

procedure Reduce(key, values):
begin
    while values.hasNext()
    begin
        individual := Individual_representation(key)
        fitness := values.getValue()
        if processed<tSize
        then
          {Wait for individuals to join in the tournament and put
               them for the last rounds}
            tournamentArray [tSize + processed%tSize] := individual
        else
          {Conduct a tournament over the past window}
            SelectionAndCrossover()
        processed := processed + 1

          {Finished all reduces}
        if processed_all_individuals
        then
          {Cleanup for the last tournament windows}
            for k=1 to tSize
            begin
                SelectionAndCrossover()
                processed := processed + 1
            end
    end
end

procedure SelectionAndCrossover:
begin
    crossoverArray[processed%cSize] := Tournament(tournamentArray)
    if (processed-tSize)%cSize = cSize - 1
    then
          {Perform crossover whenever the crossover window is full}
        newIndividuals := Crossover(crossoverArray)
        for individual in newIndividuals
          Emit(individual, dummyFitness)
end
```

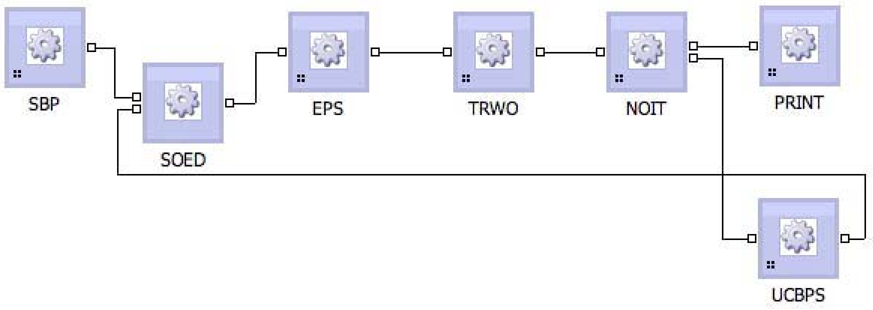


Fig. 3. Meandre's flow for the proposed selectorecombinative genetic algorithm flow

creates the stream of individuals that forms a population, and the second one updates the fitness of the individuals as they are streamed by. The *recombination* components—as introduced in section 2.2—will require and internal state. As individuals stream by, it requires two individuals in order to perform the uniform crossover operation.

The *selection* component requires a bit more thinking. One may think that a similar implementation to the one used for the *recombination* component may work approximately accurate enough. However, such an implementation would be equivalent to implement a spatial based selection method instead of a tournament one. Spatially constraint selection methods have been shown to elongate the takeover time, and thus reduce the selection pressure when compared to tournament selection without replacement [9, 38, 39, 29, 15]. Also, following on the temptation of accumulate all the individuals and then recreate the stream as we conduct tournaments against the accumulated population also seems prone to introduce a large sequential bottleneck and, thus, leaving the execution profile prone to the Amdahl's law [2]. The answer is simpler. Create all the required tournaments when you get the *stream initiator*. Then as individuals are streamed in perform the possible tournaments and start streaming the new selected population. Thus, we will guarantee that as individuals are still streamed in, we are already streaming out of the component a newly selected population, minimizing Ambdahl's law impact.

Figure 3 presents the components discussed above and how they get assembled to for the final data-intensive flow. The `sbp` (stream binary population) streams a population of individuals to start the flow. Individuals get streamed into the `soed` (single open entry door) component. This is a special component. Its only goal is to make sure that all the individuals on the initial population are streamed into the evaluation `eps` component before the next streamed population arrives. The goal of this component is to avoid having individuals from the next population mixed with the previous population ones. It is important to note here that individuals my still be streamed in from the initialization when new individuals are already being streamed out the recombination `ucbps` component, and hence, we must guarantee that the two populations do not get mixed. Evaluated individuals are then streamed into the tournament selection `twrops` component, and the selected individuals are

Listing 4. Portion of the ZigZag script implementing a selectorecombinative genetic algorithm using Meandre components.

```
#
# Main import and aliases
#

# Omitted...

#
# Component instances
#
sbp, soed, eps = SBP(), SOED(), EPS()
noit, twrops, ucbps = NOIT(), TWROPS(), UCBPS()
print = PRINT()
#
# The flow
#
@new_pop, @cross_pop = sbp(), ucbps()
@pop_ed = soed(
                initial_stream:new_pop.population;
                stream:cross_pop.population
             )
@eval_pop = eps(population:pop_ed.stream)
@sel_pop = twrops(population:eval_pop.population)
@cnt = noit(population:sel_pop.population)
ucbps(population:cnt.population)
print(population:cnt.final_population)
```

streamed into the `noit` (number of iterations) component which allows a population stream to go by a given number of iterations and then diverts the population to a secondary output port to print the final output to the console. If the finalization criteria is not met, the selected individuals are streamed into the recombination `ucbps` component. New offspring will be sent for evaluation passing through the `soed` safe gate. Program 4 presents the ZigZag script implementing this flow. A detailed description on how to implement Meandre components and write ZigZag scripts can be found elsewhere—see [30] and `http://seasr.org/meandre`.

4.2 The Compact Genetic Algorithm and Hadoop

We encapsulate each iteration of the CGA as a seperate single MapReduce job. The client accepts the commandline parameters, creates the initial probability vector splits and submits the MapReduce job. Let the probability vector be $P = \{p_i : p_i = Probability\_of\_the\_variable(i) = 1\}$. Such an approach would allow us to scale over a billion variables, if P is partitioned into m different partitions $P_1, P_2, \ldots, P_m$ where m is the number of mappers.

Listing 5. Map phase of each iteration of the CGA

```
procedure Map(key, value):
begin
    splitNo := key
    probSplitArray := value
    Emit(splitNo, [0, probSplitArray])
    for k := 1$ to tournamentSize
    begin
        individual := nil
        ones := 0
        for prob in probSplitArray
        begin
            if Random(0,1) < prob
            then
                individual := 1
                ones := ones + 1
            else
                individual := 0
            Emit(splitNo, [k, individual])
            WritetoDFS(k, ones)
        end
    end
end
```

Map. Generation of the two individuals matches the MAP function, which has to be computed independent of other instances. As shown in the algorithm in Listing 5, the MAP takes a probability split P_i as input and outputs the *tournamentSize* individuals splits, as well as the probability split. Also, it keeps track of the number of ones in both the individuals and writes it to a global file in the Distributed File System (HDFS). All the reducers, later read these values.

Reduce: We implement Tournament selection without replacement. A tournament is conducted among *tournamentSize* generated individuals and the winner and the loser is selected. Then, the probability vector split is updated accordingly. A detailed description of the reduce step can be found on Listing 6.

Optimizations: We use optimizations similar to the After initial experimentation, we noticed that for larger problem sizes, the serial initialization of the population takes a long time. Similar to the optimizations used while MapReducing SGAs, we create the initial population in a seperate MapReduce phase, in which the MAP generates the initial probability vector and the REDUCE is the Identity Reducer.

The bits of the variables in the individual are compactly represented in an array of **long long int**s and we use efficient bit operations for crossover and fitness calculations. Also, we use **long long int**s to represent probabilities instead of floating point numbers and use the more efficient integer operations.

Listing 6. Reduce phase of each iteration of the CGA

```
procedure Initialize:
begin
    Allocate_and_initialize(OnesArray[tournamentSize])
    winner := −1
    loser := −1
    processed := 0
    n := 0
    for k:=1 to tournamentSize
    begin
        for r=1 to numReducers
        do
            Ones[k] := Ones[k] + ReadFromDFS(r, k)
            if Ones[k] > winner
            then
                winnerIndex := k
            else
                if Ones[k] < loser
                then
                    loserIndex := k
    end
end

procedure Reduce(key, values):
    while values.hasNext()
    begin
        splitNo := $key
        value[processed] := values.getValue()
        processed := processed + 1
    end
    for prob in value[0]
    begin
        if value[winner].bit[n] != value[winner][n]
        then
            if value[winner].bit[n] = 1
            then
                newProbSplit [n] := value[0] + 1/population
            else
                newProbSplit [n] := value[0] − 1/population
    end
    Emit(splitNo, [0, newProbSplit])
end
```

4.3 The Extended Compact Genetic Algorithm and Meandre

As we did with the selectorecombinative genetic algorithm, and loosely following Foster's methodology [13], we will identify what data is going to drive our execution. In this particular case, the relevant pieces of information used by eCGA's model building are the gene partitions used to compute the MPM. The greedy model-building algorithm requires exploring a large number of possible partition merges while building the model—being $\mathscr{O}(\ell^3)$ the worst case scenario. Thus, this would suggest that the partitions of genes should be the basic piece of data to stream. At each step of the model building process, a stream of partitions will need to be evaluated to compute each combined complexity score. The evaluation of each partition is also independent of each other, further simplifying the overall design.

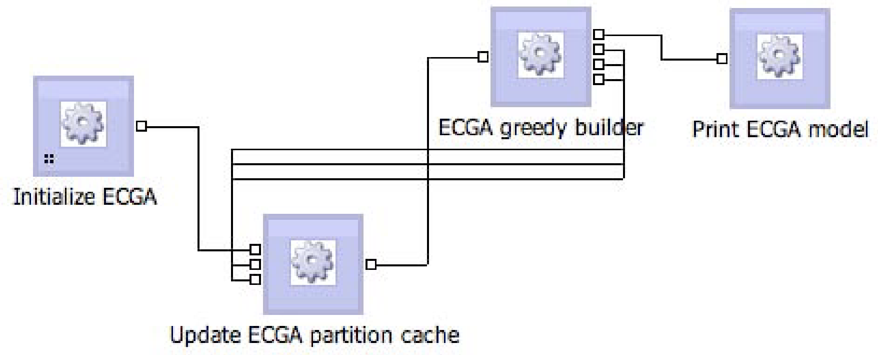


Fig. 4. Meandre's flow for eCGA

Figure 4 presents the four components we will use to implement a data-intensive version of eCGA model builder. The `init_ecga` component creates a new population, evaluates the individuals (using and MK deceptive trap [17] where $k = 4$ and $d = 0.25$), pushes the selected population obtained using tournament selection without replacement ($s = 6$), and starts streaming the initial set of gene partitions that require evaluation. Then, the `update_partition` component computes the combine complexity of the partition and streams that information to the `greedy_ecga_mb` component. This component implements the greedy algorithm that receives the evaluated partitions and decides which ones to merge. In case that a partition merge is possible, the new set of partitions to be evaluated are streamed into the `update_partition` component. If no merger is possible, the `greedy_ecga_mb` component pushes the final MPM model to the `print_model` component. Program 7 present the ZigZag script implementing the data-intensive computing version of eCGA.

Listing 7. ZigZag script implementing the extended compact genetic algorithm using Meandre components.

```
#
# Main import and aliases
#

# Omitted...

#
# Component Instances
#
init_ecga, greedy_ecga_mb = INIT_ECGA(), GREEDY_ECGA_MB()
update_partitions = UPDATE_ECGA_PARTITIONS()
print_model, print_pop = PRINT_MODEL(), PRINT_POP_MATRIX()
#
# The flow
#
@init_ecga = init_ecga()
@update_part = update_partitions(
      ecga_partition_cache:
            init_ecga.ecga_partition_cache;
      ecga_partition_to_update_i:
            init_ecga.ecga_partition_to_update_i;
      ecga_partition_to_update_j:
            init_ecga.ecga_partition_to_update_j
     )
@greedy_mb = greedy_ecga_mb(
      ecga_partition_cache:
            update_part.ecga_partition_cache
      )
update_partitions(
      ecga_partition_cache:
            greedy_mb.ecga_partition_cache;
      ecga_partition_to_update_i:
            greedy_mb.ecga_partition_to_update_i;
      ecga_partition_to_update_j:
            greedy_mb.ecga_partition_to_update_j
      )
print_model(ecga_model:greedy_mb.ecga_model)
```

5 Experiments

This section presents the results obtained using the Hadoop and Meandre frameworks to scale the proposed GAs and EDAs. The section begins presenting the results achieved using both frameworks to speedup traditional genetic algorithms.

Then it reviews the results obtained using Hadoop to speedup cGA—more fitted to tackle large data-volume problems width relatively easy to implement algorithms. Finally the section concludes presenting the promising scalability results achieved using Meandre on eCGA being, to the best of our knowledge, one of the first attempts that has succeeded in showing that efficient parallelization of eCGA model building is possible.

5.1 Selectorecombinative Genetic Algorithms

To illustrate the benefits of both frameworks, we implemented and tested the selectorecombinative genetic algorithm following the descriptions presented above.

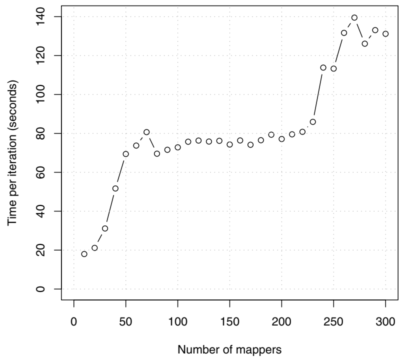
5.1.1 Hadoop and SGAs

We implemented the simple ONEMAX problem on Hadoop (0.19)[11] and ran it on our 416 core (52 nodes) Hadoop cluster. Each node runs a two dual Intel Quad cores, 16GB RAM and 2TB hard disks. The nodes are integrated into a Distributed File System (HDFS) yielding a potential single image storage space of $2 \cdot 52/3 = 34.6TB$ (since the replication factor of HDFS is set to 3). A detailed description can be found elsewhere[12]. Each node can run 5 mappers and 3 reducers in parallel. Some of the nodes, despite being fully functional, may be slowed down due to disk contention, network traffic, or extreme computation loads. Speculative execution is used to run the jobs assigned to these slow nodes, on idle nodes in parallel. Whichever node finished first, writes the output and the other speculated jobs are killed. For each experiment, the population for the GA is set to $n \log n$ where n is the number of variables.

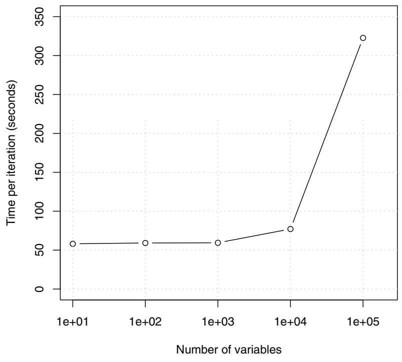
We ran two sets of experiments. In the first one, we kept the load set to 1,000 variables per mapper. As shown in Figure 5(a), the time per iteration increases initially and then stabililizes around 75 seconds. Thus, increasing the problem size as more resources are added does not change the iteration time. Since, each node can run a maximum of 5 mappers, the overall map capacity is $5 \cdot 52(nodes) = 260$. Hence, around 250 mappers, the time per iteration increases due to the lack of resources to accommodate so many mappers. In the second set of experiments,we utilize the maximum resources and increase the number of variables. As shown in Figure 7(b), our implementation scales to $n = 10^5$ variables, keeping the population set to $n \log n$. Adding more nodes would enable us to scale to larger problem sizes. The time per iteration increases sharply as the number of variables is increased to $n = 10^5$ as the population increases super-linearly ($n \log n$), which is more than 16 million individuals.

[11] http://hadoop.apache.org

[12] http://cloud.cs.illinois.edu



(a) Scalability of selectorecombinative genetic algorithm with constant load per node for the ONEMAX problem.



(b) Scalability of selectorecombinative genetic algorithm for ONEMAX problem with increasing number of variables.

Fig. 5. Results obtained using Hadoop when implementing a simple selectorecombinative genetic algorithm

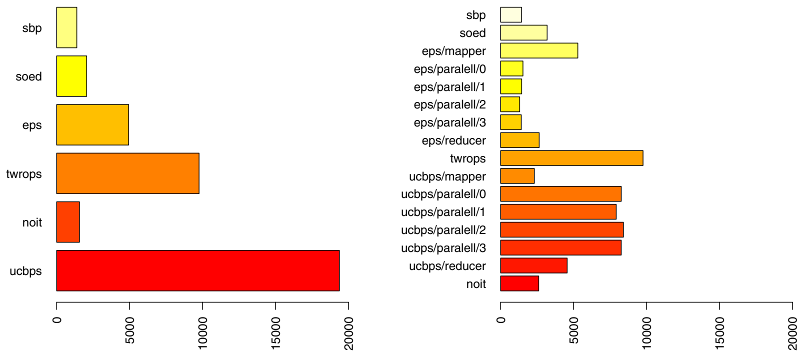
5.1.2 Meandre and SGAs

We run some experiments to illustrate the properties of data-intensive computing modeling. Unless noted otherwise, the experiments were run on an Intel 2.8GHz Quad Core equipped with 4Gb of RAM, running Ubuntu Linux 8.0.4, and Sun JVM 1.5.0_15. The problem we solved was a relatively small OneMax [16, 17], for $5,000$ bits and a population size of $10,000$—details on the population sizing can be found elsewhere [17]. The goal of the experiment was to reveal the execution profile of the converted algorithm. Figure 6(a) presents the average time spent by each component. Times are averaged over 20 runs, all evolving the optimal solution. The first thing to point out is that evaluation is not the most expensive part of the execution. OneMax is so simple, that the cost of selection and crossover dominates the execution.

Such counter intuitive profile would be a problem is we took a traditional parallelization route based on master/slave configurations delegating evaluations [8]—which works its best on the presence on costly evaluation functions. Thanks to choosing a data-intensive computing—and Meandre's ability to automatically parallelize components[13]—we can also automatically parallelize the costly part of the execution: the uniform crossover[14]. Also, we can, at the same time parallelize the evaluation, which in this situation may have little effect. However, the key property to highlight is that either in this cases, or in the case of having a costly evaluation function, the underlying data-intensive flow algorithm does not need to be changed, and component parallelization will help, for a given problem, parallelize the costly

[13] Multiple instance of a component can be spawned to process in parallel incoming individuals.

[14] Same considerations would apply in case of having a mutation component.



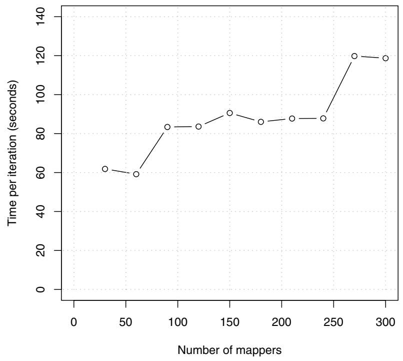
(a) Original (b) Parallelized

Fig. 6. Execution profile of the original data-instensive flow implementing a selectorecombinative genetic algorithm and its automatically parallelized version of `epb` and `ucbps` components (parallelization degree equal to the number of available cores, 4). Times are in milliseconds and are averages over twenty runs.

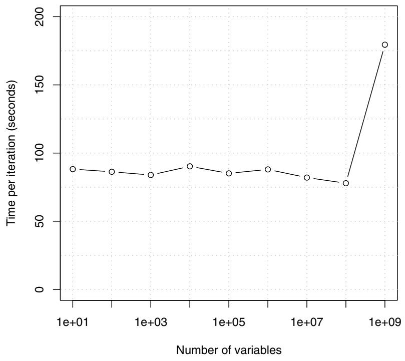
parts of the execution profile—see Figure 6(b). Hence, the inherent nature of data-intensive computing can help focus attention where is really needed. Also, parallelization, can introduce new bottlenecks—see `twrops` times on Figure 6(b)—, which now we could also parallelize to make such bottleneck disappear. Next section will show the scalability benefits that this data-intensive approach can help unleash.

5.2 The Compact Genetic Algorithm and Hadoop

To better understand the behavior of the Hadoop implementation of cGA, we repeated the two experiment sets done in the case of the Hadoop SGA implementation. For each experiment, the population for the cGA is set to $n \log n$ where n is the number of variables. As done previously, first we keep the load set to 200,000 variables per mapper. As shown in Figure 7(a), the time per iteration increases initially and then stabililizes around 75 seconds. Thus, increasing the problem size as more resources are added does not change the iteration time. Since, each node can run a maximum of 5 mappers, the overall map capacity is $5 \cdot 52(nodes) = 260$. Hence, around 250 mappers, the time per iteration increases due to the fact that no available resources (mapper slots) in the Hadoop framework are available. Thus, the execution must wait till mapper slots are released and the remaining portions can be executed, and the whole execution completed.



(a) Scalability of compact genetic algorithm with constant load per node for the ONEMAX problem.



(b) Scalability of compact genetic algorithm for ONEMAX problem with increasing number of variables.

Fig. 7. Results obtained using Hadoop when implementing a the compact genetic algorithm.

In the second set of experiments, we utilized the maximum resources and increase the number of variables. As shown in Figure 7(b), our implementation scales to $n = 10^8$ variables, keeping the population set to $n \log n$.

5.3 *The Extended Compact Genetic Algorithm and Meandre*

We ran three different experiments. First we measure the executions profile of the implement data-intensive eCGA. Figure 9(a) presents the average time spend in each component over 20 runs—$\ell = 256$ and $n = 100,000$. All the runs lead to learning the optimal MPM model thanks to the oversized population. Figure 9(a) highlights the already known fact that most of the execution time of the model building process is spent evaluating the gene partitions. Also, the initialization is being negatively affected by the partition updates, since it is being held back since it produces partitions much faster than they can be evaluated. This fact can be observed on Figure 9(b). After providing four parallelized partition evaluation components, not only the overall wall clock time drop, but also the initialization time too, since now, the update input queues can keep up with the initial set of partitions generated.

We repeated these experiments providing $\{1,2,3,$ and $4\}$ parallelized `update_partition` components and measured the overall speedup against the traditional eCGA model building implementation. Figure 8 presents the speedup results graphically. The first element to highlight is that, only using one `update_partition` component instance we obtained a superlinear speedup. This is the result of few improvements on the implementation of the components, which allowed to

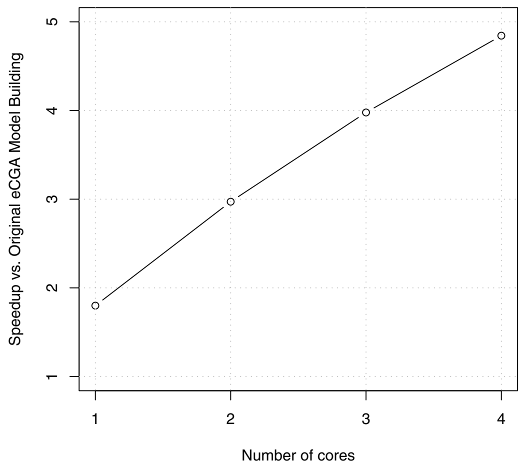


Fig. 8. eCGA speedup compared to the original non data-intensive computing implementation. Figure show the speedup as a function of the number of `update_partitions` parallelized components available.

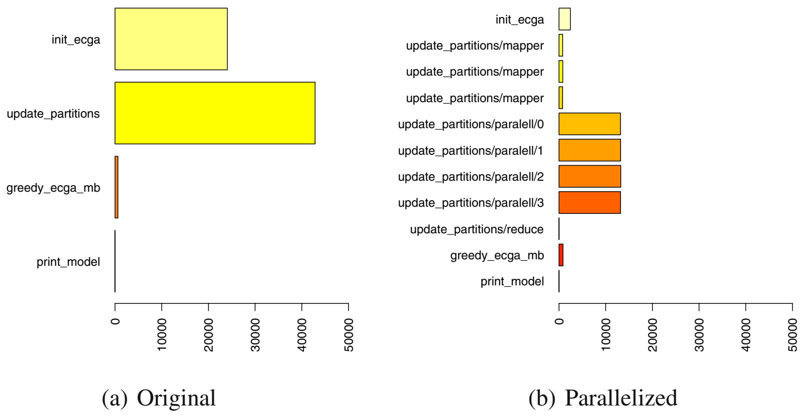


Fig. 9. Execution profile of the original data-instensive flow implementing eCGA and its automatically parallelized version of its `update_partitions` component (parallelization degree equal to the number of available cores, 4). Times are in milliseconds and are averages of twenty runs.

remove extra layers of unnecessary function calls present in the original code. Also, the fact that partitions results are streamed into the greedy model builder adds and extra improvement—similar to pipeline segmentation as discussed earlier and the availability of idle cores[15]—by advancing computations instead of waiting for the last partition to be calculated. The final speedup graph shows a clear linear increase in performance as more cores are efficiently used despite resources contention when using all the cores available. Finally, we ran the same experiment on a SGI Altix machine with a multiprocessor NUMA architecture at the National Center for Supercomputing Applications (NCSA)[16] and requested 16 and 32 nodes. The averaged speedup was computed over 20 independent runs. Again, the speedup showed a linear speedup of 14.01 and 27.96 of 16 and 32 processors. The slight drop on performance is the results of memory contention of the NUMA interconnection architecture of the SGI Altix machine.

6 Related Work

Several different models like fine grained [31], coarse grained [26] and distributed [25] models have been proposed for implementing parallel GAs. Traditionally, MPI has been used for implementing parallel GAs. However, MPIs do not scale well on commodity clusters where failure is the norm, not the exception. Generally, if a node in an MPI cluster fails, the whole program is restarted. In a large cluster, a machine is likely to fail during the execution of a long running program, and hence fault tolerance is necessary. MapReduce [11] is a programming model that enables the users to easily develop large-scale distributed applications. Hadoop[17] is an open source implementation of the MapReduce model. Several different implementations of MapReduce have been developed for other architectures: Phoenix [37] for multicores, CGL-MapReduce [12] for streaming applications.

To the best of our knowledge, MRPGA [23] is the only attempt at combining MapReduce and GAs. However, they claim that GAs cannot be directly expressed by MapReduce, extend the model to MapReduceReduce and offer their own implementation. We point out several shortcomings: Firstly, the Map function performs the fitness evaluation and the "ReduceReduce" does the local and global selection. However, the bulk of the work - mutation, crossover, evaluation of the convergence criteria and scheduling is carried out by a single co-ordinator. As shown by their results, this approach does not scale above 32 nodes due to the inherent serial component. Secondly, the "extension" that they propose can readily be implemented within the traditional MapReduce model. The local reduce is equivalent to and can be implemented within a Combiner [11]. Finally, in their **mapper**, **reducer** and **final_reducer** functions, they emit "*default_key*" and 1 as their values. Thus, they

[15] The hardware used for this experiment did not provide a fair way to execute the data-intensive flow using only one core. If that could have been possible, a normal linear speedup curve would have been obtained when extra cores were added and the time of executing on one core used to compute the speedup instead of the time of the original sequential implementation.

[16] http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/SGIAltix/TechSummary/

[17] http://hadoop.apache.org

do not use any characteristic of the MapReduce model - the grouping by keys or the shuffling. The Mappers and Reducers might as well be independently executing processes only communicating with the co-ordinator.

We take a different approach, trying to hammer the GAs to fit into the MapReduce model, rather than change the MapReduce model itself. We implement GAs in Hadoop, which is increasingly becoming the de-facto standard MapReduce implementation and used in several production environments in the industry.

7 Conclusion

In this paper we have shown that implementing evolutionary computation algorithms using a data-intensive computing paradigms is possible. We have presented step-by-step transformations for three illustrative cases—selectorecombinative genetic algorithms and estimation of distribution algorithms—and reviewed some best practices during the process. Transformations have shown that either Hadoop's MapReduce model, or Meandre's semantic-driven data-intensive flows can help scale easily and transparently evolutionary computation algorithms. Moreover, our results have also shown the inherent benefits of the underlying usage of data-intensive computing frameworks and how, when properly engineered, these algorithms can directly benefit from the current race on increasing the number of cores per chips without having to change the original data-intensive flow.

Results have shown that Hadoop is an excellent choice when we have to deal with large problems, as long as resources are available, being able to maintain iteration times relatively constant despite the problem size. We have also shown that using Meandre linear speedups are possible without changing the underlying algorithms based on data-intensive computing thanks to the its inherent parallel processing. We have also shown that such results hold for multicore architectures, but also for multiprocessor NUMA architectures.

We are current exploring how the extended compact genetic algorithm could be implemented using a MapReduce paradigm, as well as finishing Meandre's implementation of the compact genetic algorithm. Our future work is focused on analyzing other evolutionary computation algorithms that may display different execution profiles than the ones used in this book chapter, and what challenges they may face when developing their data-intensive computing counterparts.

Acknowledgments

This work was funded, in part, by NSF IIS Grant #0841765. We would also like to thank The Andrew W. Mellon Foundation and the National Center for Supercomputing Applications for their funding and support to the SEASR project that has made possible the development of the Meandre infrastructure.

References

1. Alba, E. (ed.): Parallel Metaheuristics. Wiley, Chichester (2007)
2. Amdahl, G.: Validity of the single processor approach to achieving large-scale computing capabilities. In: AFIPS Conference Proceedings, pp. 483–485 (1967)
3. Baluja, S.: Population-based incremental learning: A method of integrating genetic search based function optimization and competitive learning. Tech. Rep. CMU-CS-94-163, Carnegie Mellon University (1994)
4. Baluja, S., Caruana, R.: Removing the genetics from the standard genetic algorithm. Tech. Rep. CMU-CS-95-141, Carnegie Mellon University (1995)
5. Beckett, D.: RDF/XM Syntax Specification (Revised). W3C Recommendation 10 February 2004, The World Wide Web Consortium (2004)
6. Beynon, M.D., Kurc, T., Sussman, A., Saltz, J.: Design of a framework for data-intensive wide-area applications. In: HCW 2000: Proceedings of the 9th Heterogeneous Computing Workshop, p. 116. IEEE Computer Society, Washington (2000)
7. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation 10 February 2004, The World Wide Web Consortium (2004)
8. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Springer, Heidelberg (2000)
9. De Jong, K., Sarma, J.: On decentralizing selection algorithms. In: Proceedings of the Sixth International Conference on Genetic Algorithms, pp. 17–23. Morgan Kaufmann, San Francisco (1995)
10. de la Ossa, L., Sastry, K., Lobo, F.G.: Extended compact genetic algorithm in C++: Version 1.1. IlliGAL Report No. 2006013, University of Illinois at Urbana-Champaign, Urbana, IL (2006)
11. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI 2004: Sixth Symposium on Operating System Design and Implementation (2004)
12. Ekanayake, J., Pallickara, S., Fox, G.: Mapreduce for data intensive scientific analyses. In: ESCIENCE 2008: Proceedings of the 2008 Fourth IEEE International Conference on eScience, pp. 277–284. IEEE Computer Society, Washington (2008), http://dx.doi.org/10.1109/eScience.2008.59
13. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley, Reading (1995)
14. Foster, I.: The virtual data grid: A new model and architecture for data-intensive collaboration. In: 15th International Conference on Scientific and Statistical Database Management, p. 11 (2003)
15. Giacobini, M., Tomassini, M., Tettamanzi, A.: Takeover time curves in random and small-world structured populations. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, pp. 1333–1340. ACM, New York (2005), http://doi.acm.org/10.1145/1068009.1068224
16. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, MA (1989)
17. Goldberg, D.E.: The Design of Innovation: Lessons from and for Competent Genetic Algorithms. Kluwer Academic Publishers, Norwell (2002)
18. Goldberg, D.E., Deb, K., Clark, J.H.: Genetic algorithms, noise, and the sizing of populations. Complex Systems 6, 333–362 (1992); (Also IlliGAL Report No. 91010)
19. Goldberg, D.E., Korb, B., Deb, K.: Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems 3(5), 493–530 (1989)

20. Harik, G., Cantú-Paz, E., Goldberg, D.E., Miller, B.L.: The gambler's ruin problem, genetic algorithms, and the sizing of populations. Evolutionary Computation 7(3), 231–253 (1999); (Also IlliGAL Report No. 96004)
21. Harik, G., Lobo, F., Goldberg, D.E.: The compact genetic algorithm. In: Proceedings of the IEEE International Conference on Evolutionary Computation, pp. 523–528 (1998); (Also IlliGAL Report No. 97006)
22. Harik, G.R., Lobo, F.G., Sastry, K.: Linkage learning via probabilistic modeling in the ECGA. In: Pelikan, M., Sastry, K., Cantú-Paz, E. (eds.) Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications, ch. 3. Springer, Berlin (in press) (Also IlliGAL Report No. 99010)
23. Jin, C., Vecchiola, C., Buyya, R.: MRPGA: An extension of mapreduce for parellelizing genetic algorithms. In: Press, I. (ed.) IEEE Fouth International Conference on eScience 2008, pp. 214–221 (2008)
24. Larrañaga, P., Lozano, J.A. (eds.): Estimation of Distribution Algorithms. Kluwer Academic Publishers, Boston (2002)
25. Lim, D., Ong, Y.S., Jin, Y., Sendhoff, B., Lee, B.S.: Efficient hierarchical parallel genetic algorithms using grid computing. Future Gener. Comput. Syst. 23(4), 658–670 (2007), http://dx.doi.org/10.1016/j.future.2006.10.008
26. Lin, S.C., Punch, W.F., Goodman, E.D.: Coarse-grain parallel genetic algorithms: Categorization and new approach. In: Proceeedings of the Sixth IEEE Symposium on Parallel and Distributed Processing, pp. 28–37 (1994)
27. Llorà, X.: E2K: evolution to knowledge. SIGEVOlution 1(3), 10–17 (2006), http://doi.acm.org/10.1145/1181964.1181966
28. Llorà, X.: Data-intensive computing for competent genetic algorithms: A pilot study using meandre. In: Proceedings of the 2009 conference on Genetic and evolutionary computation (GECCO 2009). ACM Press, Montreal (in press, 2009)
29. Llorà, X.: Genetic Based Machine Learning using Fine-grained Parallelism for Data Mining. Ph.D. thesis, Enginyeria i Arquitectura La Salle. Ramon Llull University, Barcelona (February 2002)
30. Llorà, X., Ács, B., Auvil, L., Capitanu, B., Welge, M., Goldberg, D.E.: Meandre: Semantic-driven data-intensive flows in the clouds. In: Proceedings of the 4th IEEE International Conference on e-Science, pp. 238–245. IEEE Press, Los Alamitos (2008)
31. Maruyama, T., Hirose, T., Konagaya, A.: A fine-grained parallel genetic algorithm for distributed parallel systems. In: Proceedings of the 5th International Conference on Genetic Algorithms, pp. 184–190. Morgan Kaufmann Publishers Inc., San Francisco (1993)
32. Mattmann, C.A., Crichton, D.J., Medvidovic, N., Hughes, S.: A software architecture-based framework for highly distributed and data intensive scientific applications. In: ICSE 2006: Proceedings of the 28th international conference on Software engineering, pp. 721–730. ACM, New York (2006), http://doi.acm.org/10.1145/1134285.1134400
33. Morrison, J.P.: Flow-Based Programming: A New Approach to Application Development. Van Nostrand Reinhold (1994)
34. Mühlenbein, H.: The equation for response to selection and its use for prediction. Evolutionary Computation 5(3), 303–346 (1997)
35. Mühlenbein, H., Paaß, G.: From recombination of genes to the estimation of distributions I. Binary parameters. In: Ebeling, W., Rechenberg, I., Voigt, H.-M., Schwefel, H.-P. (eds.) PPSN 1996. LNCS, vol. 1141, pp. 178–187. Springer, Heidelberg (1996)
36. Pelikan, M., Lobo, F., Goldberg, D.E.: A survey of optimization by building and using probabilistic models. Computational Optimization and Applications 21, 5–20 (2002); (Also IlliGAL Report No. 99018)

37. Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multicore and multiprocessors systems. In: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (2007)
38. Sarma, J., De Jong, K.: An analysis of local selection algorithms in a spatially structured evolutionary algorithm. In: Proceedings of the Seventh International Conference on Genetic Algorithms, pp. 181–186. Morgan Kaufmann, San Francisco (1997)
39. Sarma, J., De Jong, K.: Selection pressure and performance in spatially distributed evolutionary algorithms. In: Proceedings of the World Congress on Computatinal Intelligence, pp. 553–557. IEEE Press, Los Alamitos (1998)
40. Sastry, K., Goldberg, D.E.: Designing competent mutation operators via probabilistic model building of neighborhoods. In: Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 114–125 (2004); Also IlliGAL Report No. 2004006
41. Sastry, K., Goldberg, D.E., Llorà, X.: Towards billion-bit optimization via a parallel estimation of distribution algorithm. In: GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation, pp. 577–584. ACM Press, New York (2007), http://doi.acm.org/10.1145/1276958.1277077
42. Sywerda, G.: Uniform crossover in genetic algorithms. In: Proceedings of the third international conference on Genetic algorithms, pp. 2–9. Morgan Kaufmann Publishers Inc., San Francisco (1989)
43. Uysal, M., Kurc, T.M., Sussman, A., Saltz, J.: A performance prediction framework for data intensive applications on large scale parallel machines. In: O'Hallaron, D.R. (ed.) LCR 1998. LNCS, vol. 1511, pp. 243–258. Springer, Heidelberg (1998)
44. Weibel, S., Kunze, J., Lagoze, C., Wolf, M.: Dublin Core Metadata for Resource Discovery. Tech. Rep. RFC2413, The Dublin Core Metadata Initiative (2008)
45. Welge, M., Auvil, L., Shirk, A., Bushell, C., Bajcsy, P., Cai, D., Redman, T., Clutter, D., Aydt, R., Tcheng, D.: Data to Knowledge (D2K). Tech. rep., Technical Report Automated Learning Group, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign (2003)

Evolvable Agents: A Framework for Peer-to-Peer Evolutionary Algorithms

Juan Luis Jimenez Laredo, Juan Julian Merelo Guervos,
and Pedro Angel Castillo Valdivieso

Abstract. Distributed evolutionary computation programs often needs increasingly big amounts of computational power when tackling large instances of hard optimization problems, and Peer-to-Peer (P2P) systems could be an option for building the large virtual supercomputer in which they could be run. Even as distributed Evolutionary Algorithms (EA) do take advantage of parallel execution by simultaneously promoting diversity and reducing runtime, there are still many challenges on the parallelization of EAs in P2P systems. In this chapter we present a survey of the state of the art in P2P EAs and our solutions to the main P2P issues such as decentralization, massive scalability and fault tolerance.

1 Introduction

The aim of this chapter is to provide a general overview on the issues, properties and performance of Peer-to-Peer (P2P) Evolutionary Algorithms (EAs). As in any other distributed EA, a P2P EA has got two different facets, the algorithmic and the computational performance. The first is related to the structural changes that the algorithm suffers when deployed on several processors while the latter corresponds to the computational speedup that can be expected. Indeed, distributed EAs are studied as a way of preserving genetic diversity while improving the runtime of the algorithm [5]. To this end, a good understanding on the physical platform, P2P systems in this case, benefits an adequate design.

From a general point of view, P2P systems are application level networks (ALN) able to constitute a single virtual computer composed of a potentially large number

Juan Luis Jimenez Laredo · Juan Julian Merelo Guervos · Pedro Angel Castillo Valdivieso
Department of Architecture and Computer Technology, University of Granada, Spain
e-mail: {juanlu,jmerelo,pedro}@geneura.ugr.es

of interconnected resources. The computational power is provided by a group of users connected to the Internet who share their spare CPU cycles (e.g. the BOINC project is a successful case of virtual supercomputer based on volunteers sharing their computers' CPU cycles [2]). However, there are still many challenging issues in the parallelization of EAs in P2P systems. Questions such as *decentralization* (such a computation paradigm is devoid of any central server), *scalability* (since P2P systems are large-scale networks) or *fault tolerance* (given that resources are added and eliminated dynamically, often as a consequence of a decision from an user that volunteers CPUs under his control) become of the maximum interest and have to be addressed.

P2P systems define a rich set of topologies for the interconnection of nodes at application level, so-called overlay networks [25]. In parallel, within the EC area, spatially structured EAs focus on the study of different topologies as population structure for an EA (see [28] for a survey). Hence, a distributed P2P EA can be designed as a spatially structured EA in which the population structure is defined by a P2P overlay network.

Additionally to the standard approaches in EAs stating that mate choice depends just on fitness (*panmixia*), spatially structured EAs define a population in which any given individual has a restricted number of neighbours and the chances for mating are, therefore, reduced within the neighbourhood. Population structures can be modeled as a graph in which the vertices are individuals[1] and edges represent relationships between them. Since a graph can be easily mapped to a network topology, a spatially structured EA can be easily distributed.

The key to the P2P EA presented in this chapter is the Evolvable Agent model (*EvAg*) proposed by the authors in [16]. It consists of a fine grained and decentralized approach for parallelizing EAs with a population of concurrent and self-scheduled agents performing the evolutionary steps of selection, variation and evaluation of individuals. The population structure is based on a gossiping P2P protocol called *newscast* and presented in [14]. *Newscast* behaves asymptotically as a small-world graph in which every pair of nodes are connected through a short sequence of intermediate nodes [31].

An inherent advantage of using P2P protocols as population structure is that they are designed to tackle large-scale graphs and they present consequently a good scalability behavior (see the study of scalability of the newscast protocol by Voulgaris et al. in [30]). This way, results in [17] about the scalability of the *EvAg* model are specially remarkable under large instances of hard optimization problems. As a general property of these problems, the evaluation cost increases with respect to the size of the problem instance. Hence, large instances imply a bigger computational cost on the evaluation function. Additionally, the problem complexity increases with size, making the problem more difficult to solve. Resolution methods based on population, as EAs, require large population sizes in order to tackle such instances with enough reliability and cycle-sharing P2P systems are a large and mostly free source of computing power.

[1] In this chapter, we refer equally to the terms individual and node, since each individual has its own schedule and could potentially be placed in a different node.

Besides scalability, fault tolerance is also a main issue in the design of P2P applications since resources in P2P systems are prone to failure. Such systems are subject to the dynamics of peers: a node joins the system, contributes some resources and leaves it afterwards [22]. The independent arrival and departure of thousands of peers causes a collective effect called *churn*. The *EvAg* model has been tested in [18] under different *churn* conditions. In spite of the departure of nodes, possibly containing valid solutions, the *EvAg* model is able to reach the success criterion (a success rate of 0.98) in all the test-cases. Furthermore, assuming no restrictions in the amount of available peers, the runtime of the algorithm scales independently of either the *churn* scenario or the population size, which confirms that the *EvAg* model is robust and fault tolerant.

Finally, there are two major current lines of application in which P2P EAs are promising: Results in [17] show the suitability of such algorithms for tackling large instances of problems with high requirements in computing power via the massive scale-up of P2P systems. On the other hand, the *EvAg* model has been applied to dynamic optimization problems (DOP) in [20]. The P2P approach outperforms the results of the state of the art algorithm Self-Organizing Random Immigrants GA (SORIGA) presented in [27]. SORIGA adopts a Self-Organized Criticality model in order to maintain a sub-population of random individuals and their offspring which varies in size by a power-law distribution. Key to such an improvement is that, as a spatially structured EA, *EvAg* preserves genetic diversity at the inhomogeneities of the small-world relationships between individuals.

All the above mentioned issues are discussed in detail within the different sections of this chapter. Section 2 reviews the state of the art literature related to P2P EAs, from the first attempts to the current lines of research. Section 3 provides some insights into the role that the population structure plays on the environmental selection pressure of an EA, so that, in Section 4, the overall architecture of the *EvAg* model can be better understood as a spatially structured and decentralized EA. From that point, the rest of the chapter focuses on the properties of the *EvAg* model under the different issues of P2P EAs. In Section 5, a *scalability* analysis of the model is presented, in Section 6 the *fault tolerance* is assessed under different *churn* scenarios and in Section 7, the proposed P2P EA is applied to DOP. Finally, Section 8 extends the content of the chapter by exposing some conclusions and future challenges.

2 State of the Art

The idea of distributed Evolutionary Algorithms was proposed quite early (e.g. Grefenstette in [12]); nowadays, parallel EAs are approached mainly using three models: master-slave, islands and fine grained spatially structured EAs. However, P2P EAs are more recent and not all the models fit with the issues involving P2P systems, such as decentralization, massive scalability or fault tolerance, as we will see below.

- In the *master-slave* model the algorithm runs on the master node and the individuals are sent for evaluation to the slaves, in an approach usually called *farming*. Such an architecture does not match decentralized structures and the master represents a single point of failure.
- One of the most usual and widely studied approaches in parallel EAs is the *island model* (see [5] for a survey). The idea behind this model is that the global panmictic population is split in several sub-populations or demes called islands. The communication pipes between islands are defined by a given topology, through which they exchange individuals (migrants) with a certain rate and frequency. The migration will follow a selection policy in the source island and a replacement policy in the target one. Practitioners use to establish a fixed population size P in scale-up studies, a number of islands N and a population size per island of P/n where $n = 1, \dots, N$. The work by Hidalgo and Fernández [13] requires a special attention. They experimentally show how the algorithmic results are strongly dependent on the number of islands. Our experimentation in [19] with the Island model is consistent with such a conclusion since it shows to be very sensitive to parameter calibration and P2P systems do not provide a priori knowledge of the global environment that an island model would need in order to set parameters such as the number of islands, the population size per island and the migration rate.
- Most of the works regarding *finer grained approaches* for parallel EAs focus on the algorithmic effects of using different topologies i.e. Giacobini et al. study the impact of different neighbourhood structures on the selection pressure in regular lattices [7] and different graph structures such as a toroid [8] or small-world [10]. This last structure has shown empirically to be competitive against panmictic EAs [23, 9]. Fine grained approaches are more suitable for decentralization as stated in [32, 15], where the key underlying idea is that individuals evolve within a defined set of neighbours. Following this line, we presented in [16] a formal model for P2P EAs, it is the *Evolvable Agent model* that we analyse in this chapter. The model uses the gossiping protocol newscast [14] as population structure. Newscast was proposed within the DREAM project [4], one of the pioneers in distributed P2P EAs. Although, the island-based parallelization of DREAM was shown in [21] to be insufficient for tackling large-scale decentralized scenarios.

On the other hand, protocols such as newscast have been designed taking into account both EAs and P2P architecture: as a gossiping protocol, newscast is scalable and robust [30]. There are evidences that such properties extend to P2P EAs using newscast as population structure. The scalability analysis in [17] shows that the *EvAg* model is able to tackle large instances of hard optimization problems in reasonable time (i.e. the runtime scales with fractional power with respect to the problem instance). Additionally, the study of fault tolerance in [18] shows that, whenever a P2P system guarantees enough peers at the beginning of an experiment, the departure of nodes does not inflict a penalization either on the convergence nor on the runtime. Such a study uses the model of *churn* proposed by Stutzbach and Rejaie in [26]. Indeed, Fernández de la Vega in [29] advanced that Evolutionary Algorithms are fault tolerant because of their nature and design.

3 Population Structure as a Complex Network

To help understand the role of the population structure in a P2P EA, we introduce the structural design of a simple and easy understandable complex network proposed by Watts and Strogatz in [31]. As described by the authors, the procedure for building a small-world topology can start from a ring lattice with n vertices and k edges per vertex. With a given probability p, each edge is rewired at random. This way for a rewiring factor of $p = 0$ the ring lattice is kept while for $p = 1$ a random graph is generated. It is within the intermediate values of p where the graph gets small-world structure.

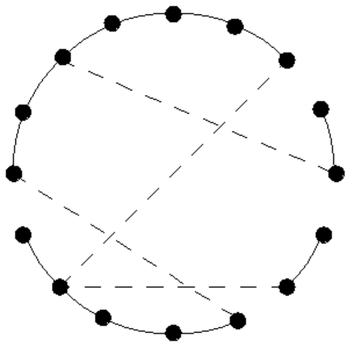


Fig. 1. Small-world graph built from a ring lattice with $n = 16$, $k = 2$ and $p = 0.2$. Being the Average Path Length (APL) the average shortest distance between any pair of nodes, this graph has APL = 3.7, the original ring lattice had APL = 4.26 while a complete graph (i.e. panmictic) would have APL = 1.0.

Figure 1 shows a small-world topology built from a ring lattice. Despite having a larger average path length than panmictic graphs, the inhomogeneity in such kind of topologies was shown in [10] to induce qualitatively similar selection pressures on EAs than panmictic population structures.

The influence in the environmental selection pressure of such population structures can be represented by their takeover time curves. Goldberg and Deb in [11] define the takeover time as the time that takes for a single, best individual to take over the entire population without any other mechanism than selection. Hence, takeover time is the proportion of best individuals as a function of time.

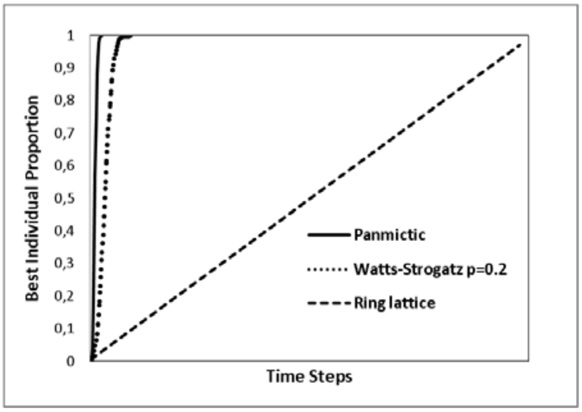


Fig. 2. Takeover time curves in a panmictic graph, Watts-Strogatz graph with $p = 0.2$ and the original ring lattice with $k = 2$. All the results for $n = 1600$ and binary tournament.

Figure 2 shows that the takeover time curve in the Watts-Strogatz graph is similar to a panmictic graph meaning that the induced selection pressures with both topologies are roughly equivalent. As in Watts-Strogatz small-world topologies, this chapter shows that P2P topologies can induce similar selection pressures to the panmictic one, allowing in addition a better scalability behaviour at the lower edge cardinality of P2P systems.

4 Overall Model Description

The overall procedure of our approach consists of a population of Evolvable Agents (*EvAg*), described in Section 4.1, whose main design objective is to carry out the main steps of evolutionary computation: selection, variation and evaluation of individuals [6]. Each *EvAg* is a node within a newscast topology in which the edges define its neighborhood. For the sake of simplicity, we assume a newscast node as a peer. However, a peer could hold several nodes in practice.

4.1 Evolvable Agent

An *EvAg* itself is an EA composed of a single individual [15, 16]. In spite of the model not having a population in the canonical sense, adjacent *EvAgs* provide each other with the genetic material that they require to evolve. Therefore, we talk about a population of *EvAgs* instead of a population of individuals.

Algorithm 1 shows the pseudo-code of an *EvAg* where the agent owns an evolving solution (S_t).

Algorithm 1. Evolvable Agent

```
$S_t \Leftarrow$ Initialize Agent
**loop**
    Sols $\Leftarrow$ Local Selection(Newscast) *See algorithm 2*
    $S_{t+1} \Leftarrow$ Recombination(Sols)
    $S_{t+1} \Leftarrow$ Mutation($S_{t+1}$)
    Evaluate($S_{t+1}$)
    **if** $S_{t+1}$ better than $S_t$ **then**
        $S_t \Leftarrow S_{t+1}$
    **end if**
**end loop**
```

The selection takes place locally into a given neighborhood where each agent selects the current solutions (S_t) of adjacent agents. Selected solutions are stored in *Sols* ready to be recombined. Within this process a new solution S_{t+1} is generated. If the newly generated solution S_{t+1} is better than the old one S_t, it replaces the current solution.

4.2 A Note on Local Performance

Locally to a computer, every *EvAg* is scheduled within a thread and dispatched by the operating system. The multi-threading nature of the model implies an impact on the local throughtput, expressed as:

$$Throughput_{EA} = \frac{Computational \quad Effort}{Time} \tag{1}$$

where Computational Effort is usually understood in EC as the number of fitness evaluations.

Either the context exchange of the threads or the mutual exclusion mechanisms have a computational cost which is avoid in sequential approaches.

Nevertheless, Symmetric Multiprocessing (SMP) architectures are becoming nowadays very popular as desktop machines and sequential approaches are unable to take advantage of more than a single processor. This way, the local performance of the *EvAg* model can be assessed by measuring the speed-up in the throughput with respect to a sequential GA (sGA):

$$Speed-up = \frac{Throughput_{EvAg}}{Throughput_{sGA}} \tag{2}$$

To this end, the computational cost of the evaluation function (i.e. the independent variable in the throughput equation) is scaled from few milliseconds to one second.

Figure 3 shows that the throughput speeds-up asymptotically, having a limit on the number of processors. Therefore, the performance in single processor machines tends to be equivalent to sequential approaches as the evaluation cost increases while is clearly outperformed in SMP machines. An additional advantage of the *EvAg* model is that the load balance is transparent for the programmer since it is carried out by the operating system.

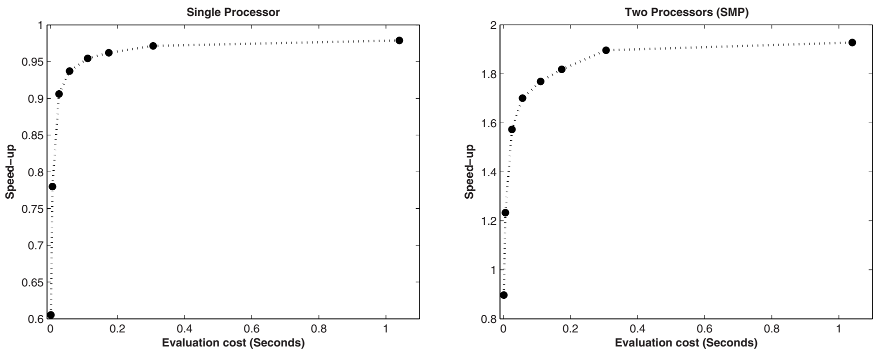


Fig. 3. The figure depicts how the EvAg throughput speeds up with respect to the one yielded by sGA when the evaluation function cost scales for a population size of 400 individuals. The test-bed is a single processor (*left*) and a dual-core processor (*right*).

4.3 Population Structure

In principle, the *EvAg* model places no restrictions in the choice of population structure, although this choice will have an impact on the dynamics of the algorithm since it establishes the environmental selection pressure. As it has been previously said, we apply the newscast protocol to create the population structure.

Algorithm 2 shows the pseudo-code of the main tasks in the self-organized process which builds the newscast graph. Each node maintains a cache with one entry per node in the network at most. Each entry provides information about a foreign node: A time-stamp of the entry creation (it allows the replacement of old items), and an agent identifier.

There are two different tasks that the algorithm carries out within each node. The active thread which initiates communication and the passive thread that waits for the exchange of information. In addition, the local selection procedure provides the *EvAg* with other agents' current solutions ($EvAg_h(S_t)$ and $EvAg_k(S_t)$). After ΔT time each $EvAg_i$ initiates a communication process (active thread). It selects randomly an $EvAg_j$ from $Cache_i$ with uniform probability. Both $EvAg_i$ and $EvAg_j$ exchange their caches and merge them following an aggregation function. In our case, the aggregation consists of picking up the newest items (newscast) for each cache entry in $Cache_i$, $Cache_j$ and merging them into a single cache that $EvAg_i$ and $EvAg_j$ will share. We have fixed ΔT to once per evaluation.

Algorithm 2. Newscast protocol in node $EvAg_i$

```
Active Thread
while $EvAg_i$ not finished do
    sleep $\Delta T$
    $EvAg_j \Leftarrow$ Random selected node from $Cache_i$
    send $Cache_i$ to $EvAg_j$
    receive $Cache_j$ from $EvAg_j$
    $Cache_i \Leftarrow$ Aggregate ($Cache_i$,$Cache_j$)
end while

Passive Thread
while $EvAg_i$ not finished do
    wait $Cache_j$ from $EvAg_j$
    send $Cache_i$ to $EvAg_j$
    $Cache_i \Leftarrow$ Aggregate ($Cache_i$,$Cache_j$)
end while

Local Selection(Newscast)
$[EvAg_h, EvAg_k] \Leftarrow$ Random selected nodes from $Cache_i$
```

The cache size plays an important role in the newscast algorithm. It represents the maximum number of connections (edges) that a node could have. For example, a topology with n nodes and a cache size of n, will lead to a panmictic graph topology. Therefore, the cache size is smaller than the number of nodes (typically around $log(n)$) in order to get small-world features such as a small average path length and a high clustering coefficient (for further details on the dynamics refer to [30]).

Figure 4 compares the takeover time curves for a panmictic and two different parametrized newscast graphs. As explained in the previous section, similar curves denote equivalent selection pressures induced by both kind of topologies. Nevertheless, the node degree in panmictic graphs is $n-1$ while the average degree in newscast is approximately $2c$ pointing out a better scalability of the small-world approach given that $c \ll n$. Within the rest of the sections, the cache size has been fixed to 20 based on the study of performance for different cache sizes in [16].

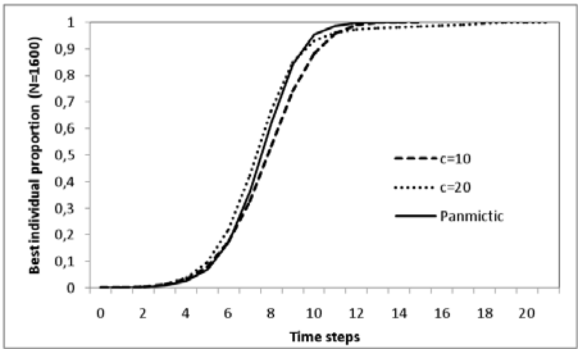


Fig. 4. Takeover time curves for a panmictic graph and two newscast graphs with $c = 10$ and $c = 20$. Results are averaged from 50 independent runs for $n = 1600$ and binary tournament.

5 Scalability

To investigate how *EvAg* scales on landscapes of different characteristics, experiments were conducted on trap functions [1]. A trap function is a piecewise-linear function defined on unitation (the number of ones in a binary string). There are two distinct regions in search space, one leading to a global optimum and the other leading to the local optimum (see Figure 5). In general, a trap function is defined by the following equation:

$$trap(u(\overrightarrow{x})) = \begin{cases} \frac{a}{z}(z - u(\overrightarrow{x})), if \quad u(\overrightarrow{x}) \leq z \\ \\ \frac{b}{l-z}(u(\overrightarrow{x}) - z), \quad otherwise \end{cases} \tag{3}$$

where $u(\overrightarrow{x})$ is the unitation function, a is the local optimum, b is the global optimum, l is the problem size and z is a slope-change location separating the attraction basin of the two optima.

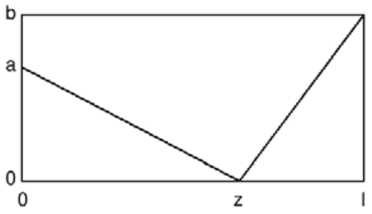


Fig. 5. Generalized *l-trap* function

For the following experiments, 2-trap, 3-trap and 4-trap functions were designed with the following parameter values: $a = l - 1$; $b = l$; $z = l - 1$. With these settings, 2-trap is not deceptive, 4-trap is deceptive and 3-trap lies in the region between deception and non-deception. Under these conditions, it is possible not only to examine how *EvAg* scales on trap functions, but also to investigate how the scalability varies when changing from non-deceptive to deceptive search landscapes. Scalability tests were performed by juxtaposing m trap functions and summing the fitness of each sub-function to obtain the total fitness.

The bisection method [24] was used for each trap and each size m to determine the optimal *EvAg* population size P, that is, the lowest P for which 98% of the runs solve the traps functions. To find it, mutation rate is set to 0, so as to search a minimum population size such that using random initialization it is able to provide enough building blocks to converge to the optimum without other mechanism than recombination and selection.

Algorithm 3 depicts the method based on bisection. The method begins with a small population size which is doubled until the algorithm ensures a reliable convergence. We define the reliability criterion as the convergence of the algorithm to the optimum 49 out of 50 times (0.98 of Success Rate). After that, the interval (min, max) is halved several times and the population size adjusted within such a range. *min* and *max* stand respectively for the minimum and maximum population size estimated.

Algorithm 3. Method based on Bisection

```
P = Initial Population Size
while Algorithm reliability < 98% do
    min = P ; max, P = Double (P)
end while
while $\frac{max-min}{min} > \frac{1}{16}$ do
    $P = \frac{max+min}{2}$
    (Algorithm reliability < 98%) ? min = P : max = P
end while
```

EvAg was tested with $p_c = 1.0$, uniform crossover and binary tournament. The standard GA stands for a 1-elitism generational GA and was used as a baseline for comparison, to this end, both aproaches were equally parameterized.

From the graphics in figure 6 it can be concluded that *EvAg* scales better than a standard GA on 2, 3 and 4-trap, but the improvement is much more noticeable when solving the deceptive 4-trap function. Under these conditions (4-trap), a standard GA faces extreme difficulties because lower order building blocks mislead the search towards local optima instead of combining to form higher order building-blocks, thus challenging the GA's search mechanisms. A possible explanation for *EvAg*'s better scalability lies in its ability to maintain genetic diversity at a higher and consequent reduction of its optimal population size (P). With a lower optimal P, *EvAg* needs fewer evaluations to reach the optimum, when compared to standard GAs.

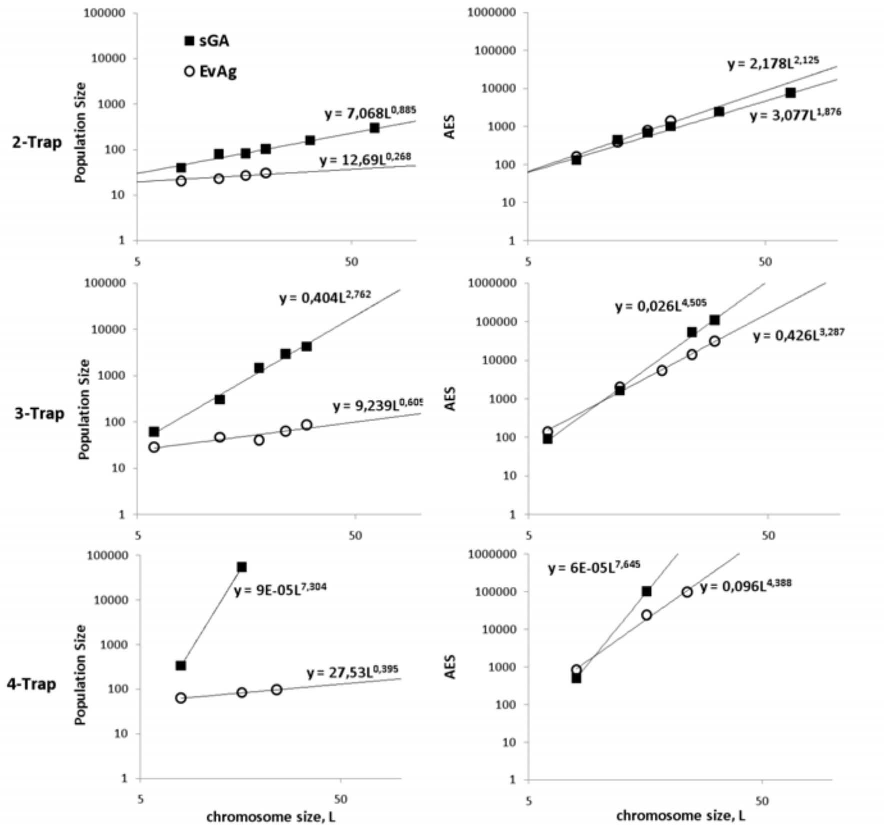


Fig. 6. Scalability with trap functions. Optimal population size and Average Evaluations to Solution (AES) values for a standard GA (sGA) and the Evolvable Agent Model (EvAg).

6 Fault Tolerance

P2P systems are large networks of volatile resources in which the collective dynamics of the peers are known as *churn*. Therefore, addressing *churn* in a P2P EA turns into a requirement of design rather than a mere study of fault tolerance.

Following the work by Stutzbach and Rejaie in [26], there are two main group-level properties of *churn* which characterize the behaviour of all participating peers: The *inter-arrival time* and the *session length*, respectively, the time between two sessions and the time from the beginning to the end of a session.

In this study we have assumed that all nodes start at the same time with a certain *session length*. The *session length* can be modeled randomly from a Weibull distribution using the following formula:

$$X = \lambda(-ln(U))^{\frac{1}{s}} \tag{4}$$

where U is drawn from the uniform distribution, s stands for the shape and λ for the scale of the Weibull distribution. The analysis by Stutzbach and Rejaie exposes that the *session length* of different P2P systems fit with a shape of $s \approx 0.40$ but the values of λ differ. Additionally, the simulator driven experiments define the time unit as a simulator cycle which could apply for different time metrics in real time. Hence, we setup the following values for $\lambda = 1, 5, 10, \infty$ depicted in Figure 7. It shows the complementary cumulative distribution functions (CCDF), representing the percentage of remaining nodes at each moment of the experiment for the different values of λ (e.g. in the cycle 10, ~8% of the peers remain for $\lambda = 1$ and ~50% for $\lambda = 10$).

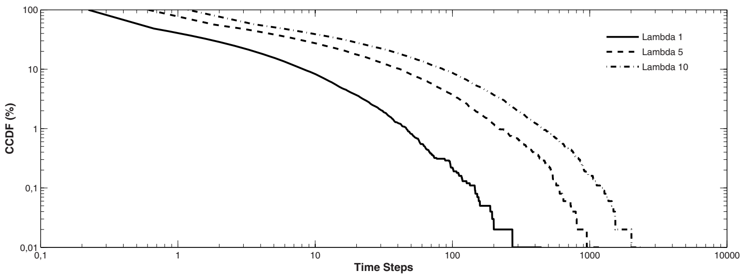


Fig. 7. Complementary cumulative distribution functions

In spite of the initial assumption, the set of *churn* scenarios allows a worst case analysis in which the system loses peers to zero. Once that a node leaves the experiment, it does not rejoin again (i.e. there is no *inter-arrival*).

Under these conditions, several instances of the 4-trap function (i.e. $L = 8, 16, 32, 64$) have been used in order to assess the impact of *churn* in the *EvAg* model. Hence, there are three variables that affect the Success Rate (SR) of the algorithm: The size of the problem (L), which will conduct to a scalability analysis, the intensity of *churn* (λ) and the population size (P).

Any of the variables have the following influence on the SR if we assume a fixed value for the rest of them. In the case of the problem instance, the bigger the size, the lower the SR. With respect to *churn*, the more departures of nodes, the lower the SR. Finally, the bigger the population size, the higher the SR.

Being λ and L two independent variables under the condition of obtaining a SR of 0.98, the population size can be expressed as a function $f(\lambda, L) = P$ and empirically estimated using the bisection method. Furthermore, the runtime of the algorithm is also analysed as a function of the three variables $g(\lambda, L, P)$ since one of the goals in any distributed EA is to reduce the time for obtaining a solution.

Figure 8 shows the scalability of the population size (P) as different curves of L, that is, L scales and λ remains fixed in $f(\lambda, L) = P$. The scalability of $f(\lambda, L)$ fits with a complexity order of $O(L^{(1.0,1.1)})$ in any of the scenarios. Hence, *churn* does not damage the scalability order (i.e. the curves are just shifted by a constant

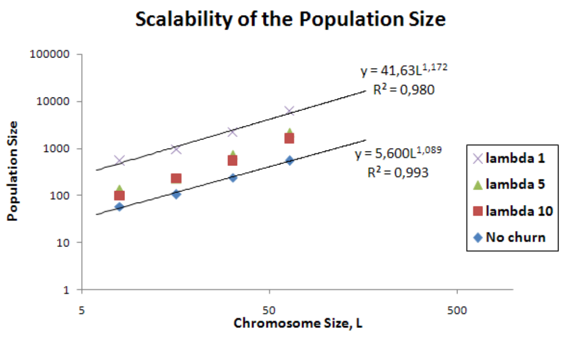


Fig. 8. Scalability of the population size $f(\lambda,L) = P$ for the 4-trap in 4 scenarios of churn (λ)

which is *churn* dependent) and a reliable convergence can be guaranteed by ensuring enough resources. This fact points to the robustness of the Evolvable Agent model.

Additionally, Figure 9 shows the runtime of the algorithm $g(\lambda,L,P)$. g is independent from λ, with an order $O(L^{0.6})$. In this case, *churn* does not affect the scalability (neither shifting the curves with a constant) and the runtime is dependent on the problem instance L. That means that we can expect the same runtime under any *churn* scenario if we ensure enough resources to satisfy the 0.98 of SR condition. Therefore, the algorithm is robust under *churn*.

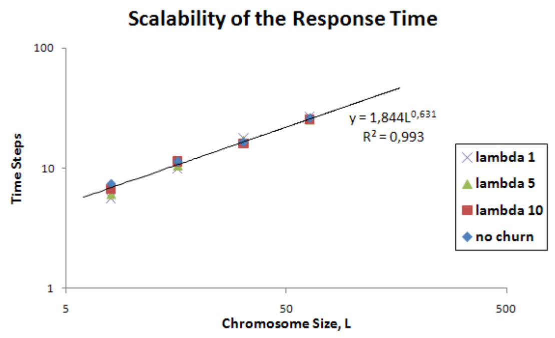


Fig. 9. Scalability of the runtime $g(\lambda,L,P)$ for the 4-trap in 4 scenarios of churn (λ) using P estimated in $f(\lambda,L)$

Figure 10 provides a better idea to the extent of these results. It represents the percentage of individuals of P for which each experiment is expected to end. The effects of *churn* are more pernicious as the instances scale. In the worst case (i.e. $L = 64$ and $\lambda = 1$), the initial population ends with a $\sim$3% of the individuals, still guaranteeing a SR of 0.98.

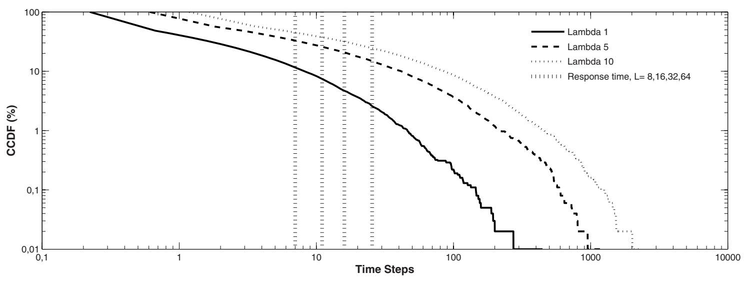


Fig. 10. Complementary cumulative distribution functions and average runtime

7 P2P EAs in Dynamic Optimization Problems

With respect to DOPs, population size P was set to 240. In order to evaluate *EvAg*'s results a standard generational GA (GGA) with 1-elitism and SORIGA were also tested with the same parameter values.

For that purpose, the DOP generator presented in [33] was used to build different changing environments based on 3-trap and 4-trap functions. Given a stationary problem ($f(x)(x \in \{0,1\}^L)$) where L is the chromosome length, DOPs may be designed by applying a binary mask to each solution before its evaluation in the following manner:

$$f(x,t) = f(x \quad XOR \quad M(k)) \tag{5}$$

where t is the generation index, k = tτ is the period index and f(x, t) is the fitness of the string x. M(k) is incrementally generated as follows:

$$M(k) = M(k-1) \quad XOR \quad T(k) \tag{6}$$

where $T(k)$ is an intermediate binary mask for every period k. T(k) has $\rho \times$L ones. ρ is a value between 0 and 1 that controls the intensity, or severity, of changes (i.e. $\rho = 0$ stands for a stationary problem and $\rho = 1$ represents the highest degree of change). Therefore, by setting ρ and τ it is possible to control two of the most important features of DOPs test environments: severity (ρ) and speed (τ) of change [3]. Nine different scenarios for each trap were designed by setting ρ to 0.05, 0.6 and 0.95, and τ to 10, 100 and 200 generations. Stationary functions were designed with 10 subfunctions each, meaning that size of dynamic 3-trap is L = 30 and size of dynamic 4-trap is L = 40. *EvAg*, GGA and SORIGA were tested with uniform crossover, bit-flip mutation, binary tournament, $p_c = 1.0$, $N = 240$ and 1-elitism (GGA). SORIGA's parameter r_r was set to 3.

GAs performance analysis on DOPs must be addressed in a different manner from static environments' usual procedure. Dynamic behaviour throughout the run must be examined, rather than the final convergence. For that purpose, the evaluation of the algorithmic performance is done by measuring the mean best-of-generation values (this is the standard procedure for DOPs). In addition, the progression of

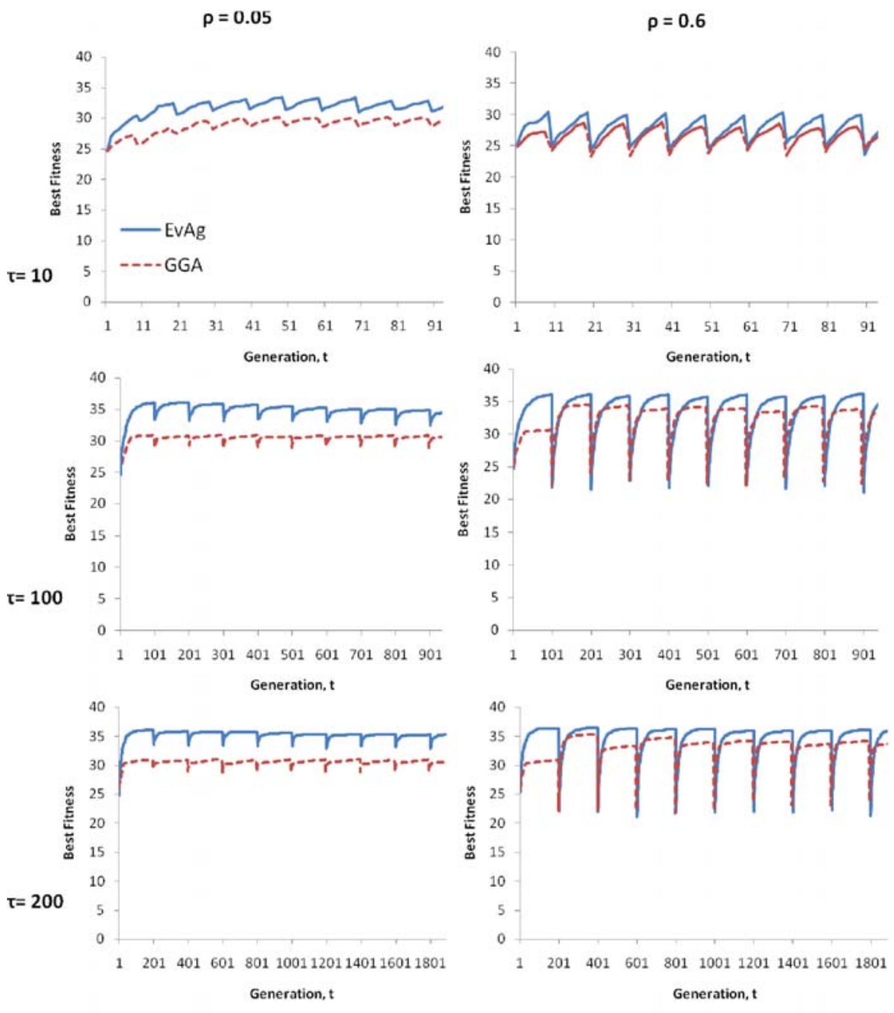


Fig. 11. Dynamics when tracking 4-trap functions ($L = 40$) for $\rho = 0.05$ and $\rho = 0.6$. *Best of generation* curves

best-of-generation values may be plotted in a graph, thus helping to understand how the algorithm reacts to changes in the environment. Different mutations rates were tested, and results in table 1 show the best configurations, that is, the mutation rates that attained the higher values when averaging the mean best-of-generation of the nine scenarios.

Table 2 helps to understand the relevance of the results in table 1 by showing the results of pairwise t-test that compares the algorithms' performance. The (+) sign means that algorithm 1 is significantly better than algorithm 2, (∼) means that the performance is equivalent and (−) means that the second GA is better. While

Table 1. Results on dynamic 3-trap and 4-trap (averaged over 30 independent runs). Mean of *best of generation* and corresponding standard deviation values.

	τ	10			100			200		
	ρ	0.05	0.6	0.95	0.05	0.6	0.95	0.05	0.6	0.95
3-trap	GGA	25.72	21.88	24.19	29.4	26	25.57	29.81	26.7	25.6
	$(p_m = \frac{1}{L})$	±0.97	±0.27	±0.24	±0.44	±0.33	±0.17	±0.11	±0.33	±0.22
$L = 30$	SORIGA	**26.43**	22.74	24.10	**29.74**	**27.71**	26.67	**29.86**	**28.8**	27.95
	$(p_m = \frac{1}{2L})$	±0.62	±0.3	±0.26	±0.05	±0.17	±0.16	±0.02	±0.1	±0.16
	EvAg	25.71	**22.83**	**26.37**	28.91	27.04	**27.83**	29.61	27.64	**28.03**
	$(p_m = \frac{1}{L})$	±0.38	±0.43	±0.34	±0.26	±0.17	±0.39	±0.27	±0.2	±0.33
4-trap	GGA	28.92	26.63	31.66	30.52	32.55	35.1	30.61	33.08	35.3
	$(p_m = \frac{1}{L})$	±0.33	±0.39	±0.52	±0.57	±0.34	±0.11	±0.44	±0.29	±0.129
$L = 40$	SORIGA	28.64	26.54	29.4	34.12	32.4	34.7	**35.92**	33.43	35.02
	$(p_m = \frac{1}{8L})$	±0.71	±0.36	±0.55	±1.57	±0.37	±0.14	±1.4	±0.38	±0.09
	EvAg	**31.71**	**27.82**	**32.71**	**34.89**	**33.76**	**36.9**	35.32	**34.87**	**37.17**
	$(p_m = \frac{1}{L})$	±0.53	±0.39	±0.46	±0.5	±0.28	±0.31	±0.51	±0.25	±0.33

Table 2. Pairwise *t-test* on dynamic 3-trap and 4-trap. Evolvable Agent vs. GGA and SORIGA.

t-test	τ	10			100			200		
	ρ	0.05	0.6	0.95	0.05	0.6	0.95	0.05	0.6	0.95
3-trap	EvAg vs. GGA	$\sim$	$+$	$+$	$-$	$+$	$+$	$-$	$+$	$+$
	EvAg vs. SORIGA	$-$	$\sim$	$+$	$-$	$-$	$+$	$-$	$-$	$+$
4-trap	EvAg vs. GGA	$+$	$+$	$+$	$+$	$+$	$+$	$+$	$+$	$+$
	EvAg vs. SORIGA	$+$	$+$	$+$	$+$	$+$	$+$	$+-$	$+$	$+$

in 3-traps GGA and SORIGA still outperform *EvAg* in some scenarios, in 4-traps our proposal achieves better results, with statistical significance, in all the scenarios except one. *EvAg* abilities to solve DOPs appear to emerge when facing a harder problem for GAs.

Figure 11 shows the dynamic behaviour of *EvAg* and GGA throughout the run. It is clear that *EvAg* is more able to track the optimum, maintain a lower distance to the solution during the search, in all scenarios.

8 Conclusions

This chapter reviews the main issues in distributed EAs over P2P systems such as decentralization, scalability or fault tolerance, the state of the art solutions to deal with them and some promising fields of application as DOPs.

To that end, the *EvAg* model has been presented and assessed empirically under different scenarios using trap-functions as a benchmark. Given that trap-functions have been designed to be difficult for EAs, the results should be easily extended to more general discrete or combinatorial optimization problems.

The evolution takes place among a population of *EvAgs* in which the population structure is managed by the gossiping protocol newscast. The population size scales with a complexity order of $O(L^{(1.0,1.1)})$ which demands for a big amount of resources. Besides, the expected runtime scales with fractional order which makes the algorithm efficient. Investigating scalability is of extreme importance when changing from a "toy problem" test environment to real-world problems which may require very large chromosomes to codify the solutions. Additionally, the approach shows to be robust under *churn*, once that an adequate population size guarantees a reliable convergence to the optimum, the departure of nodes does not inflict a penalization in the runtime.

Finally, this kind of topology preserves well the genetic diversity by relaxing the environmental selection pressure at the small-world relationships between individuals. This might be one of the reasons for the good results in DOPs. The *EvAg* model responds to changes outperforming standard GAs and SORIGA, one of the state of the art algorithms in DOPs.

In order to deploy the algorithm in real P2P platforms, the future challenges of P2P EAs focus on engineering issues rather than design ones. For instance, the impact of the latency between peers will have to be assessed on the run-time performance, it might be expected that the idle processing time decreases as the problem instances scale (i.e. bigger instances require a bigger computational time while the communication time can be assumed as fixed). Additionally, the bootstrapping of the system (i.e. the way an experiment is spread from a single peer to the rest of the system) or the fact that P2P systems are composed of heterogeneous nodes will have to be taken into account.

References

1. Ackley, D.H.: A connectionist machine for genetic hillclimbing. Kluwer Academic Publishers, Norwell (1987)
2. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: an experiment in public-resource computing. Commun. ACM 45(11), 56–61 (2002)
3. Angeline, P.J.: Tracking extrema in dynamic environments. In: Angeline, P.J., McDonnell, J.R., Reynolds, R.G., Eberhart, R. (eds.) EP 1997. LNCS, vol. 1213. Springer, Heidelberg (1997)
4. Arenas, M.G., Collet, P., Eiben, A.E., Jelasity, M., Merelo, J.J., Paechter, B., Preuss, M., Schoenauer, M.: A framework for distributed evolutionary algorithms. In: Guervós, J.J.M., Adamidis, P.A., Beyer, H.-G., Fernández-Villacañas, J.-L., Schwefel, H.-P. (eds.) PPSN 2002. LNCS, vol. 2439, pp. 665–675. Springer, Heidelberg (2002)
5. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell (2000)
6. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer, Heidelberg (2003)

7. Giacobini, M., Tomassini, M., Tettamanzi, A., Alba, E.: Selection intensity in cellular evolutionary algorithms for regular lattices. IEEE Transactions on Evolutionary Computation 9(5), 489–505 (2005)
8. Giacobini, M., Alba, E., Tettamanzi, A., Tomassini, M.: Modeling selection intensity for toroidal cellular evolutionary algorithms. In: Deb, K., et al. (eds.) GECCO 2004. LNCS, vol. 3102, pp. 1138–1149. Springer, Heidelberg (2004)
9. Giacobini, M., Preuss, M., Tomassini, M.: Effects of scale-free and small-world topologies on binary coded self-adaptive CEA. In: Gottlieb, J., Raidl, G.R. (eds.) EvoCOP 2006. LNCS, vol. 3906, pp. 86–98. Springer, Heidelberg (2006)
10. Giacobini, M., Tomassini, M., Tettamanzi, A.: Takeover time curves in random and small-world structured populations. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, pp. 1333–1340. ACM, New York (2005)
11. Goldberg, D.E., Deb, K.: A comparative analysis of selection schemes used in genetic algorithms. In: Foundations of Genetic Algorithms, pp. 69–93. Morgan Kaufmann, San Francisco (1991)
12. Grefenstette, J.J.: Parallel adaptive algorithms for function optimization. Technical Report CS-81-19, Vanderbilt University, Computer Science Department, Nashville (1981)
13. Hidalgo, I., Fernández, F.: Balancing the computation effort in genetic algorithms. In: The 2005 IEEE Congress on Evolutionary Computation, vol. 2, pp. 1645–1652. IEEE Press, Los Alamitos (2005)
14. Jelasity, M., van Steen, M.: Large-scale newscast computing on the Internet. Technical Report IR-503, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands (October 2002)
15. Laredo, J.L.J., Eiben, E.A., Schoenauer, M., Castillo, P.A., Mora, A.M., Merelo, J.J.: Exploring selection mechanisms for an agent-based distributed evolutionary algorithm. In: GECCO 2007, pp. 2801–2808. ACM Press, New York (2007)
16. Laredo, J.L.J., Castillo, P.A., Mora, A.M., Merelo, J.J.: Exploring population structures for locally concurrent and massively parallel evolutionary algorithms. In: IEEE Congress on Evolutionary Computation (CEC 2008), WCCI 2008 Proceedings, pp. 2610–2617. IEEE Press, Hong Kong (2008)
17. Laredo, J.L.J., Eiben, A.E., van Steen, M., Castillo, P.A., Mora, A.M., Merelo, J.J.: P2P Evolutionary Algorithms: A suitable approach for tackling large instances in hard optimization problems. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 622–631. Springer, Heidelberg (2008)
18. Laredo, J.L.J., Castillo, P.A., Mora, A.M., Merelo, J.J., Fernandes, C.: Resilience to churn of a peer-to-peer evolutionary algorithm. Int. J. High Performance Systems Architecture 1(4), 260–268 (2008)
19. Laredo, J.L.J., Castillo, P.A., Mora, A.M., Merelo, J.J.: Evolvable agents, a fine grained approach for distributed evolutionary computing: walking towards the peer-to-peer computing frontiers. Soft Computing - A Fusion of Foundations, Methodologies and Applications 12(12), 1145–1156 (2008)
20. Laredo, J.L.J., Castillo, P.A., Mora, A.M., Merelo, J.J., Rosa, A., Fernandes, C.: Evolvable agents in static and dynamic optimization problems. In: Rudolph, G., Jansen, T., Lucas, S., Poloni, C., Beume, N. (eds.) PPSN 2008. LNCS, vol. 5199, pp. 488–497. Springer, Heidelberg (2008)
21. Laredo, J.L.J., Castillo, P.A., Paechter, B., Mora, A.M., Alfaro-Cid, E., Esparcia-Alcázar, A., Merelo, J.J.: Empirical validation of a gossiping communication mechanism for parallel EAs. In: Giacobini, M. (ed.) EvoWorkshops 2007. LNCS, vol. 4448, pp. 129–136. Springer, Heidelberg (2007)

22. Merelo, J.J., Castillo, P.A., Laredo, J.L.J., Mora, A.M., Prieto, A.: Asynchronous distributed genetic algorithms with Javascript and JSON. In: IEEE Congress on Evolutionary Computation (CEC 2008), WCCI 2008 Proceedings, pp. 1372–1379. IEEE Press, Hong Kong (2008)
23. Preuss, M., Lasarczyk, C.: On the importance of information speed in structured populations. In: Yao, X., Burke, E.K., Lozano, J.A., Smith, J., Merelo-Guervós, J.J., Bullinaria, J.A., Rowe, J.E., Tiňo, P., Kabán, A., Schwefel, H.-P. (eds.) PPSN 2004. LNCS, vol. 3242, pp. 91–100. Springer, Heidelberg (2004)
24. Sastry, K.: Evaluation-relaxation schemes for genetic and evolutionary algorithms. Technical Report 2002004, University of Illinois at Urbana-Champaign, Urbana, IL (2001)
25. Steinmetz, R., Wehrle, K.: What is this peer-to-peer about? In: Steinmetz, R., Wehrle, K. (eds.) Peer-to-Peer Systems and Applications. LNCS, vol. 3485, pp. 9–16. Springer, Heidelberg (2005)
26. Stutzbach, D., Rejaie, R.: Understanding churn in peer-to-peer networks. In: Proceedings of the 6th ACM SIGCOMM on Internet measurement (IMC 2006), pp. 189–202. ACM Press, New York (2006)
27. Tinós, R., Yang, S.: A self-organizing random immigrants genetic algorithm for dynamic optimization problems. Genetic Programming and Evolvable Machines 8(3), 255–286 (2007)
28. Tomassini, M.: Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time. Natural Computing Series. Springer, New York (2005)
29. Fernandez de Vega, F.: A fault tolerant optimization algorithm based on evolutionary computation. In: DEPCOS-RELCOMEX 2006: Proceedings of the International Conference on Dependability of Computer Systems, Washington, DC, USA, pp. 335–342. IEEE Computer Society, Los Alamitos (2006)
30. Voulgaris, S., Jelasity, M., van Steen, M.: A Robust and Scalable Peer-to-Peer Gossiping Protocol. In: Moro, G., Sartori, C., Singh, M.P. (eds.) AP2PC 2003. LNCS (LNAI), vol. 2872, pp. 47–58. Springer, Heidelberg (2004)
31. Watts, D.J., Strogatz, S.H.: Collective dynamics of "small-world" networks. Nature 393, 440–442 (1998)
32. Wickramasinghe, W.R.M.U.K., van Steen, M., Eiben, A.E.: Peer-to-Peer evolutionary algorithms with adaptive autonomous selection. In: GECCO 2007, pp. 1460–1467. ACM Press, New York (2007)
33. Yang, S., Yao, X.: Experimental study on population-based incremental learning algorithms for dynamic optimization problems. Soft Comput. 9(11), 815–834 (2005)

Evolutionary Algorithms on Volunteer Computing Platforms: The MilkyWay@Home Project

Nate Cole, Travis Desell, Daniel Lombraña González,
Francisco Fernández de Vega, Malik Magdon-Ismail, Heidi Newberg,
Boleslaw Szymanski, and Carlos Varela

1 Introduction

Evolutionary algorithms (EAs) require large scale computing resources when tackling real world problems. Such computational requirement is derived from inherently complex fitness evaluation functions, large numbers of individuals per generation, and the number of iterations required by EAs to converge to a satisfactory solution. Therefore, any source of computing power can significantly benefit researchers using evolutionary algorithms. We present the use of volunteer computing (VC) as a platform for harnessing the computing resources of commodity machines that are nowadays present at homes, companies and institutions. Taking into account that currently desktop machines feature significant computing resources (dual cores, gigabytes of memory, gigabit network connections, etc.), VC has become a cost-effective platform for running time consuming evolutionary algorithms in order to solve complex problems, such as finding substructure in the Milky Way Galaxy, the problem we address in detail in this chapter.

In order to tackle the complexity of evolutionary algorithms when applied to real world problems, different parallel models and computer architectures have been used in the past, for instance the parallel transputer network architecture [5] or a 10 nodes Beowulf style cluster [7] improved later to 1000

Nate Cole · Travis Desell · Malik Magdon-Ismail · Heidi Newberg ·
Boleslaw Szymanski · Carlos Varela
Rensselaer Polytechnic Institute. 110 8th St. Troy, NY 12180, USA
e-mail: astro@cs.rpi.edu

D. Lombraña González · F. Fernández de Vega
Centro Universitario de Mérida, Universidad de Extremadura.
Sta. Teresa Jornet, 38. 06800 Mérida (Badajoz), Spain
e-mail: fcofdez@unex.es,daniellg@unex.es

Pentiums nodes[1]. Nowadays, large efforts are still carried out to improve results while reducing computing time, by embodying parallel techniques within EAs (see e.g., [21, 18]).

One of the most promising technologies capable of circumventing the high computational requirements of EAs, and thus reducing the solution time of many applications is the *grid* computing paradigm [32]. *Grid computing* generally refers to the sharing of computing resources within and between organizations by harnessing the power of super computers, clusters and desktop PCs, which are geographically distributed and connected by networks. Grid nodes use a special software, called *middleware*, to coordinate distributed computations.

Two of the most used middleware frameworks in the world are Globus [22] and gLite [33]. These middleware frameworks are normally complex and focused on upmarket hardware and facilities. For this reason, other grid middleware employs commodity hardware to reduce economic investment and to handle the complexity of deployment to idle desktops, thus giving rise to *desktop grids* (DGC). Examples of DGC middleware are Condor [37], Xtremweb [19] and BOINC [2].

Two main kinds of DGC are available, *enterprise grids* and *volunteer computing grids*. Enterprise grids are typically more homogeneous and usually entail processors connected by a Local Area Network (LAN) under a single root of administrative control, albeit with potentially different administrative units below. On the other hand, volunteer computing grids (e.g., as enabled by BOINC [2]) are composed of Internet-connected processors volunteered by users worldwide, resulting in larger but more heterogeneous grids.

Desktop grids have the potential to provide significant processing power since desktop computers have become an essential working tool in any market. Companies and institutions provide commodity machines to their employees to improve their efficiency when solving their everyday tasks. The hardware specifications of those desktop machines become more powerful everyday: for example, quad cores, 4 gigas of RAM memory and up to 1 Terabyte hard disks, are not uncommon. Thus, desktops are really good candidates for running complex and time consuming computational experiments. Furthermore, if we take into account that most of those desktops are underutilized by their owners, there is a potential for a large processing and storage capability within current energy usage [4].

At the same time, researchers commonly face the opposite problem: real world problems approached with EAs require more computing resources than what researchers have at their disposal. Thus, a question naturally arises: why not to exploit those unused available desktop resources to help scientific applications? Desktop grid computing, and in particular, *volunteer computing*, provides a plausible answer to this question (for the sake of simplicity, we will employ the term *volunteer computing* (VC) from now on as an umbrella for

[1] For further details see
`http://www.genetic-programming.com/machine1000.html`

different but related terms: DGC, enterprise grids and volunteer computing, given the specific study described and the real-world application shown in this chapter).

VC is being successfully used for addressing problems related to climate prediction models [1], high energy physics [47], protein folding [44] and astronomy [3], [47], to name but a few. Frequently, projects rely completely on the computing power of volunteers, converting the users in a fundamental part of the project.

Yet, the number of projects using VC is still relatively narrow, and particularly unknown by Computational Intelligence community of researchers. In this chapter, we analyze the application of VC to real life problems addressed by means of Evolutionary Algorithms (EAs), and also possible extensions to other Computational Intelligence techniques. We show the VC technology, its cooperation with EAs for solving hard problems, and a real-life application: The Milky Way at home project.

The chapter is organized as follows: Section 2 presents related work on volunteer computing and evolutionary algorithms. Section 3 describes a specific astronomy problem to which we apply EAs on VC: finding sub-structure in our own Milky Way. Section 4 describes asynchronous versions of evolutionary algorithms, especially designed to be executed on heterogeneous, failure-prone volunteer computing environments. Section 5 discusses their implementation in the MilkyWay@Home project highlighting interesting results. Finally, we conclude with remarks and potential avenues for future work.

2 Related Work

The Berkeley Open Infrastructure for Network Computing (BOINC) [2] is the generalization of the well known SETI@Home project [3]. This volunteer computing middleware framework aggregates the computing power of idle desktop machines provided by volunteers worldwide (e.g., home computers or office workstations). One of the main advantages of using VC systems is that they provide large-scale parallel computing capabilities for specific classes of applications at a very low cost. Consequently, VC is a promising platform for running real world optimization problems solved by means of EAs.

The following sub-sections present the first VC projects, the VC technology more widely used and its relationships with Parallel EAs: possibilities of collaboration and first proposals.

2.1 Volunteer Computing

The first successful VC project was “The Great Internet Mersenne Prime Search” (GIMPS) [52]. The aim of this project is to find new mersenne primes.

The project has engaged 11,875 volunteers who provide 73,439 CPUs giving, as of April 2009, a processing power of 44.3 Teraflops.

Another successful VC project is Distributed.net [50]. The aim of this project is to win the *RSA Secret Key challenge* which consist of deciphering an RSA text with different difficulties. Distributed.net has won two of the challenges, the 56 and 64 RC5 bit encryption challenges [39, 38]. Additionally the Distributed.net team is also running a distributed project named OGR which tries to find the *Optimal Golomb Ruler* [51] for 24 or more marks. At the time of this writing, Distributed.net has found a solution for the OGR-24, 25 and 26. Currently they are trying to find a solution to the OGR of 27 marks.

Both projects, GIMPS and Distributed.net, are specialized VC projects for a given application. Thus, those systems cannot be used as a general tool for running any kind of research project. Nevertheless, GIMPS and Distributed.net are good examples of successful ad hoc VC projects.

Over the years, several general purpose VC middleware have been developed including Xtremweb [19], Condor [37] and BOINC [2] (as described above, we consider all the technology from the point of view of VC, and avoid subtle distinctions about DGC or VC technology).

From all the general purpose VC middleware frameworks, BOINC is the most used one. BOINC has the largest pool of volunteers/users around the world. As of April 2009, BOINC has $1,622,500$ users, providing $3,849,182$ processors which give $1,399.4 TeraFLOPS$ of aggregated computing power to VC applications. For this reason, BOINC is the best candidate for deploying an application using volunteer computing. BOINC volunteers are willing to collaborate with new projects of interest.

In summary, VC is a good candidate to run computationally intensive problems and could thus be employed to obtain "free" computing resources for running real world EA tackling hard problems. In reality, resources are not completely "free", since a community of volunteers needs to be continuously informed of the scientific goals and outcomes of the project. New volunteers need to be recruited, and existing volunteers need to be retained. However, the benefit of the keeping volunteers informed is the ensuing "open democratic science" where people can choose which research projects are worthy of their computing resources.

2.2 BOINC: A Volunteer Computing Technology

As explained above, BOINC has the largest user and volunteer community. Currently, there are 28 official supported projects and 23 non-verified BOINC projects [2]. These projects belong to different research fields like: astronomy, physics, chemistry, biology, medicine, cognitive science, earth and environmental science, mathematicians, games, etc.

[2] For further details see http://boinc.berkeley.edu/projects.php and http://boinc.berkeley.edu/wiki/Project_list

BOINC is a *multiplatform* and *open source* middleware that comes from the SETI@home project [3]. SETI@home aims at finding extraterrestrial intelligence by analyzing radio telescope data. SETI@home has engaged the largest community of BOINC users, when writing this chapter 944, 691 users are providing 2, 239, 695 hosts which produces a computer power equal to 515.6 TeraFLOPS.

Due to the success of SETI@home, the developers decided to create BOINC, based on the SETI@home software. The goal was to provide a general tool for developing new DGC projects based on their technology. The main features of BOINC are:

- *Project autonomy.* Each BOINC project is independent, so each project has its own servers and databases. Additionally there is no central directory or approval process for the projects.
- *Volunteer flexibility.* Volunteers can decide in which and how many projects they will take part. Volunteers also decide how their resources will be shared between different projects.
- *Flexible application framework.* Applications coded in C, C++ or Fortran can be run within BOINC with little or no modification.
- *Security.* BOINC employs digital signatures to protect clients from distributing viruses or malware.
- *Server performance and scalability.* The BOINC server is extremely efficient, so that a single mid-range server can handle and dispatch millions of jobs per day. The server architecture is also highly scalable by adding more machines and distributing the load between them.
- *Open source.* The BOINC code is released under the Lesser GNU General Public License version 3 [23].

BOINC is suitable for applications that have one or both of the following requirements:

- large computation requirements,
- storage requirements.

The main requirement for running an application within BOINC is that it is divisible into multiple sub-tasks or jobs that can be run independently. As we may foresee, EAs are perfect candidates for running projects supported by BOINC: the standard parallel evaluation of individuals could be performed on different volunteer computers.

If the project is considering to employ basically volunteer resources, the project's web site must be compelling to attract volunteers and take into account the volunteer's bandwidth connections: lots of users do not have fast upload/download speeds.

BOINC is composed by two key elements: the server and the clients. BOINC employs a master-slave architecture. In order to facilitate the communications between the clients and the server, the HTTP protocol is used and the clients start always the communications. Thanks to this approach, the clients can collaborate with science even if they are behind a firewall or a proxy –general security set up for communications on institutions like companies or universities.

The BOINC server is in charge of:

- *Hosting the scientific project experiments.* A project is composed by a binary (the algorithm or application) and some input files.
- *Creation and distribution of jobs.* In BOINC's terminology a job is called a "work unit" (WU). A WU describes how the experiment must be run by the clients (the name of the binary, the input/output files and the command line arguments).

On the other hand, the BOINC client connects to the server and asks for work (WU). The client downloads the necessary files (WU) and starts the computations. Once the results are obtained, the client uploads them to the server.

BOINC measures the contributions of volunteers with *credit*. A *credit* is a numerical measure of the work done by a given volunteer and his computers. Volunteers care so much about the obtained credit when collaborating with a project, and it is one of the leitmotif of continuing collaborating with a given project. Thus, it is very important to handle correctly the granted credit to users, as BOINC projects can grant/handle credit differently.

Volunteer computing has a main drawback: resources are not reliable. For this reason, many types of attacks are possible in BOINC: hacking the server, abuse of participant hosts, etc. From all the possible attacks, there are two which are very important:

- *Result falsification.* Attackers return incorrect results.
- *Malicious executable distribution.* Attackers break into a BOINC server and, by modifying the database and files, attempt to distribute their own executable (e.g. a virus) disguised as a BOINC application.

In order to avoid these possible attacks, BOINC provides several mechanisms to reduce the likelihood of some of the above attacks:

- *Result falsification can be reduced using replication*: a result is sent to at least two different clients to check out that the obtained result has not been forged. BOINC provides different types of replication: fuzzy, homogeneous or adaptive replication.
- *Malicious executable distribution is avoided as BOINC uses digital signatures to distribute the binaries.* The server uses two signatures, one public and the other private to sign the applications. The private signature is used to sign locally the binaries and the public signature is distributed to clients for checking the origin of the application by the clients. It is important to have the private key stored in safe storage not connected to Internet to avoid possible network break-ins.

To sum up, BOINC provides enough facilities to reduce as much as possible likelihood of attacks under a volunteer computing infrastructure. However, without the cooperation of administrators, the security could be risked if for example BOINC keys are stored in the server machine or protocols like Telnet are being used for accessing the machine.

2.3 Parallel Evolutionary Algorithms and VC

EA practitioners have found that the time to solution on a single computer is often prohibitively long. For instance, Trujillo et al. employed more than 24 hours to obtain a solution for a real world computer vision problem [54]. Times to solution can be much worse, taking up to weeks or even months. Consequently, several researchers have studied the application of parallel computing techniques and distributed computing platforms to shorten times to solution [20, 53].

Examples of these efforts are the old-fashioned Transputer platform [5], new modern frameworks such as Beagle [26], or grid based tools like Paradiseo [8]. However, there are not many real world problems that are using VC for running experiments using EAs.

Considering the VC technology, two main parallel approaches for running EAs are useful for profiting volunteer computing resources:

- *Parallel fitness evaluation.* Individuals can be distributed to be evaluated on different volunteer computers simultaneously. This is useful when the fitness evaluation time is the most time-consuming part of the algorithm.
- *Parallel execution of experiments.* When a number of runs are required for obtaining statistically significant results, different runs can be distributed on a number of computers. This model is also useful for high-throughput parameter sweep experiments.

The latest case is particularly useful when running experiments in conjunction with VC technology: no changes are required in the main EA algorithm. The algorithm is simply sent to a number of computers, with different input parameters if required.

If we focus on some of the techniques comprised within EAs, only Chávez et al. [42] presented an extension to the well-known genetic programming framework LilGP [14], that runs within a BOINC infrastructure. In this work, the experiments were a proof of concept using a couple of well-known GP problems: the ant on the Santa fe trail and the even parity 5 problem. Moreover, the experiments only used a controlled environment without the collaboration of external volunteers.

A more generic approach was presented by Lombraña et. al. [27, 35]. The idea was to run a modern and widely used framework for EAs, ECJ [36], within a BOINC infrastructure. In this case, the experiments used a real environment but only a reduced number of volunteers where engaged. Furthermore, the experiments were again complex GP benchmarks but no real world optimization problems were addressed.

Another approach was presented by Samples et al. [45]. Samples et. al. showed the feasibility of using DGC for a typical genetic programming parameter sweep application using a pool of desktop PCs. Nevertheless, the lack of a standard middleware and a genetic programming tool has kept this approach from being commonly adopted by researchers.

All the previous approaches were simple proof-of-concepts, thus, to the best of the author's knowledge, the only real-world problem that is already using VC and Evolutionary Algorithms is the MilkyWay@home project; which is described in depth within the next sections.

3 Finding Milky Way Galaxy Substructure

The Milky Way spheroid is one of the major components of the Galaxy. It occupies a roughly spherical volume with the other components (disks, bulge, bar, etc.) embedded in it. Despite its volume it produces only a small fraction of the starlight emitted by the Galaxy. The stellar spheroid is composed of primarily older and more metal poor stars that produce little light compared to the brighter, more massive, stars that form in the gas-rich components of the disk. The majority of the mass of the Milky Way exists within the spheroid as dark matter; however, the nature, distribution, and structure of this mass is unknown.

With the construction and operation of large scale surveys such as the Sloan Digital Sky Survey (SDSS), the Two Micron All Sky Survey (2MASS), and many other current and upcoming projects there is an "astronomical" amount of data to be sorted through. This huge amount of data is not only composed of photometric studies, but large numbers of stellar spectra are being taken with many surveys being operated or completed solely focused on taking a large number of incredibly accurate spectra, the Sloan Extension for Galactic Understanding and Exploration (SEGUE) for example. This increase in the amount of data, as well as the increase in the accuracy of this data has led to many discoveries involving the spheroid and substructure in the Galaxy. The Sagittarius dwarf spheroidal galaxy (Sgr dSph) and its associated tidal stream, were the first known example of a current merger event to be discovered [30, 28, 29, 58]. Since its discovery, the study of substructure has dominated the research efforts towards the spheroid. This has resulted in the discovery of several additional dwarf galaxies, tidal streams and globular clusters of the Milky Way as can be found in [41, 57, 55, 15, 40, 6, 56], and [59] among others.

For many years, the spheroid was imagined to have evolved from outward flows from the disk of the Galaxy [17], or to have formed in conjunction with the rest of the Galaxy and gradually evolved to its current state [46]. It has also long been imagined to have a smooth and continuous power law density distribution [25]. However, the advancement in technology and analysis techniques have discovered the large amount of substructure, discussed above, and has shown that at least some of the spheroid was constructed via merger events and that the spheroid was composed of debris from hierarchical structure formation [24]. A shift in the thinking of the spheroid has therefore come about, and the more substructure that is discovered the stronger the case that the spheroid is non-smooth and was constructed primarily in this manner.

3.1 Why Substructure?

Dwarf galaxies and star clusters are gravitationally bound systems, which can themselves be bound in the gravitationally potential of the Milky Way. As dwarf galaxies approach the center of the Milky Way, the differential gravitational forces can pull stars out of their bound orbits in the dwarf galaxy, and cause them to orbit the Milky Way instead. Stars with lower energy are pulled ahead of the dwarf galaxy core while stars with higher energy fall behind, resulting in the creation of long streams of stars, also known as tidal streams, that extend the longer the dwarf is bound by the Milky Way. As this disruption continues, the streams will become longer and cover more of the sky as more and more of the stars are stripped from the smaller body. Over time, all traces of the merging body will fade as the stars become more and more dispersed and become assimilated into the Milky Way spheroid. These long tidal streams of stars provide a unique opportunity for these are the only stars in which it is possible to know not only where they are and where they are going, but also where they have been. They trace out the path the merging body took as it traversed the gravitational potential of the Milky Way. In this way substructure in the spheroid can be seen as something of a cosmic fingerprint powder bringing all of the details of the spheroid itself into focus.

By studying the substructure of the spheroid, specifically that of tidal streams, it is possible to study the galactic potential. This is not done via direct measurement, but is primarily studied via simulations, which are adjusted to replicate the observational data. The more observational points with which to compare and the more accurate those points are, the more precisely the simulation can constrain the models of the Galactic potential. As previously stated, the majority of the mass of the Milky Way is within the spheroid and this is composed of dark matter. Therefore, the dominant component of the Galactic potential is provided by the spheroid and specifically the dark matter of the spheroid. Thus, by constraining the models of the Galactic potential it is possible to determine the distribution of dark matter within the Galaxy.

It is important to develop techniques that are accurate and efficient means of studying substructure, so the results may then be used to compare against the simulations. There are primarily two methods used for discovery and study of substructure: *kinematical* and *spatial*. The kinematical approach attempts to find co-moving groups of stars that can be identified by groups of stars in a similar location with common velocity. These are indicators that the stars might have come from a common structure instead of simply being part of the smooth spheroid. This approach, though potentially more powerful, is limited in that it requires a spectroscopic analysis of all stars studied in order to determine the velocities. Accurate stellar spectroscopy is significantly harder to obtain than photometry. The second approach for substructure discovery and analysis is to simply search for overdensities in the star counts within the data. This is done by looking for statistically relevant deviations from the assumed background distribution of the spheroid. This technique benefits from the fact that only a photometric analysis of the data need be accomplished, thus the amount of data available for a study of this kind is much greater. The majority of the substructure discoveries in the Milky Way spheroid have been made through analysis of the photometric data.

3.2 The Maximum Likelihood Approach to Substructure

We have developed a maximum likelihood method for the discovery and analysis of substructure within the stellar spheroid. This method seeks to provide an efficient, automated, accurate, and mathematically rigorous means to study substructure. The method has been designed to determine the spatial characteristics of tidal debris and the stellar spheroid through the use of photometric data.

The SDSS is a large, international collaboration that has generated an enormous amount of data over 10,000 square degrees of the sky. The SDSS data was taken with a 2.5m dedicated telescope at Apache Point Observatory in New Mexico. Due to its placement in the northern hemisphere, the data covers primarily the north Galactic cap with some minimal coverage in the south. The SDSS survey area is composed of 2.5° wide stripes taken on great circles across the sky (the entire sky is comprised of 144 such stripes) with data taken at high Galactic latitude. Since the SDSS imaging survey is well calibrated and is comprised of mostly contiguous data, it is a good candidate for studying substructure as it traces across the sky.

Specifically, we extract stars of the color of blue F turnoff stars in SDSS data to study substructure. The relatively large number of F turnoff stars make them a good candidate for a statistical study of this nature. Also, the color of F turnoff stars within the spheroid is bluer than that of the Milky Way disk, therefore contamination of stars from non-spheroid components can be minimized necessitating a model for only one Galaxy component, the spheroid. Finally, F turnoff stars were chosen for this study for it is possible to reasonably estimate their distance although not as well as other less numerous "standard candles." By modeling the distribution of absolute magnitudes that F turnoff stars can take, it is possible to utilize stars of this type quite effectively.

Tidal streams generally follow very complex paths across the sky and are therefore difficult to model. However, over small volume, such as a 2.5° wide stripe of data taken by the SDSS, the tidal stream may be approximated as linear. In this manner, the tidal stream is estimated in a piecewise fashion across the sky with each stripe of data maintaining its own set of stream parameters. In this way, the tidal stream density distribution is modeled as a cylinder with Gaussian fall off from its axis. The tidal stream model is thus parameterized by the cylinder position, its orientation, and the standard deviation of the Gaussian fall off. The smooth spheroid component is modeled as a standard Hernquist profile and is therefore parameterized with a scaling factor in the Galactic Z direction, and a core radius of the power law component. Finally, the absolute magnitude distribution in F turnoff stars is modeled as a Gaussian distribution with fixed variance.

Utilizing the above models we developed a likelihood function for the probability of the data given the model and the parameters. The input data set is thus composed of a three column file in which each row represents the spatial position of an F turnoff star as observed via the SDSS telescope. An optimization method is then applied to the likelihood function and the best-fit parameters are determined for the given dataset and the models. A detailed description of the model descriptions and likelihood function can be found in [11]. The approach of this algorithm offers a unique advantage in that the spheroid and tidal debris are fit simultaneously. Because of this, it becomes possible to probabilistically extract the tidal debris distribution from that of the spheroid. In this manner, the structures can then be separately analyzed via additional means. This provides a method of independent study previously unavailable to researchers.

After successfully testing the algorithm upon a simulated data set, the initial goal has been to analyze the Sgr tidal stream across all of the SDSS data. The algorithm has successfully been used to analyze the stream in the three stripes of data in the southern Galactic cap. The results presented in [11] for the analysis of stripe 82 were a success, for the well established values determined via other methods were recovered, and the error bars upon the results generated via this method are much smaller than those previously achieved. However, the true potential of this method can be seen in the analysis of the two surrounding stripes [10]. These include a new detection of the Sgr tidal stream in the south as well as the great improvement in the position of the previous detection in the opposing stripe. Figure 1 depicts the results of analyzing these three stripes and how these results compare to a model of the Sgr dSph disruption. Ongoing research seeks to create a complete map of the leading Sgr tidal stream throughout the north Galactic cap (and fill in the northern part of the figure). A preliminary analysis of part of this data has already shown significant discrepancy between the model of the Sgr dSph disruption and the spatial properties being determined via this

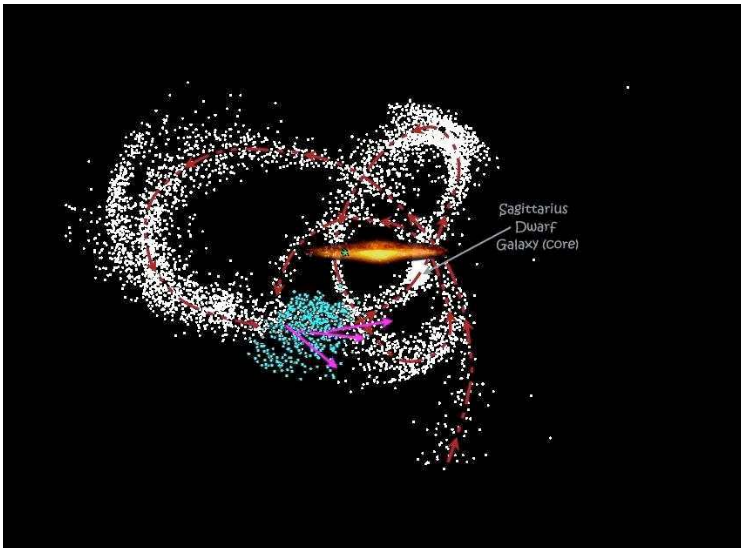


Fig. 1. Sagittarius dwarf disruption with fits to SDSS southern stripes. Plotted here is the Sgr dSph face on, which is nearly orthogonal to the plane of the Milky Way. A galaxy consistent with the Milky Way has been overlayed and the Solar position is marked with a star. The dotted arrow shows the orbital path of the dwarf and the current position of the Sgr dwarf core is clearly labeled. The points following this orbital path are a subsampling of an N-body simulation consistent with the result from [34]. The complete arrows depict the position and spatial direction of the Sgr tidal stream within the three southern SDSS stripes (from left:79, 82, and 86). The remaining points represent a subsampling of those stars found to fit the density profile of the stream within the stripe using the separation technique to extract the stream from the stellar spheroid.

method. However, the breadth of this discrepancy will not be known until a complete analysis of the data has been performed. It is also quite interesting to note that in order to accurately fit the data in the northern stripes, it is necessary to fit multiple tidal streams within the same dataset, for there is substructure beyond the Sgr tidal stream in the data collected from the north. Therefore, even though the analysis of the Sgr tidal stream is the primary concern of the project at this stage, an analysis of other substructure will be occurring simultaneously.

Following the completion of the Sgr tidal stream study, we would like to use the information obtained to attempt to constrain the models for the Galactic potential by fitting a simulation of the Sgr dSph to the values obtained in this study. This will be difficult, for an appropriate measure of the "goodness of fit" must be determined with which to compare the simulation and the analysis results. Also, the computational complexity of an undertaking of this nature is far beyond that of even the current algorithm. This is due to not only having to analyze the data, but having to actually create the data via N-body simulation. Another topic of great interest that will provide a significant challenge, is an effort to determine the true distribution of the smooth spheroid component. To do this, all substructure must be removed from the data via this algorithm and then all remaining data (representing the smooth spheroid) fit at once. This will demand significant changes to the current algorithm to fit over multiple datasets at once, plus a method for determining the correct model must be determined as well.

4 Asynchronous Search

Maximum likelihood estimation attempts to find the set of parameters of a mathematical function that best fits a given data set. The mathematical function represents a scientific model, for example, the geometrical substructure of the stars in a galaxy wedge, given certain parameters such as a star stream position, width, intersection angle, etc. The data set represents the "ground truth", for example, the observed photometric galaxy data, or the result of an N-body simulation.

Population based search methods such as differential evolution (DE) [48], genetic algorithms (GA) [9], and particle swarm optimization (PSO) [31, 16] use the notion of *generations* of individuals in a population that evolves over time. Analogous to their biological counterparts, the most *fit* individuals have better probability of survival and reproduction in newer generations. We call *genetic search* the process of using genetic algorithms to search in n-dimensional space for the best set of parameters that optimizes a given n-arity function. In genetic search, individuals are sets of parameters to a function, and their fitness value is computed by applying the function to the parameter set. For this work, individuals are sets of real valued numbers, which are the input parameters to the maximum likelihood calculation.

Traditional population based search methods, such as differential evolution, genetic search and particle swarm optimization are typically iterative in nature, which limits their scalability by the size of the population. This section introduces an asynchronous search (AS) strategy, which while being similar to traditional population based search methods in that it keeps a population of parameters (or individuals) and uses combinations and modifications of those individuals to evolve the population. The main difference is that AS uses a master-worker approach instead of a parallel model of concurrency. Rather than iteratively generating new populations, new members of the population are generated in response to requests for work and the population is updated whenever work is reported to the master.

Asynchronous search consists of two phases and uses two asynchronous message handlers. The server can either be processing a *request work* or a *report work* message and cannot process multiple messages at the same time. Workers repeatedly request work then report work. In some ways this approach is very similar to steady-state genetic search, where n members of the population are replaced at a time by newly generated members.

In the first phase of the algorithm (while the population size is less than the maximum population size) the server is being initialized and a random population is generated. When a *request work* message is processed, a random parameter set is generated, and when a *report work* message is processed, the parameters and the fitness of that evaluation are added to the server's population. When enough *report work* messages have been processed, the algorithm proceeds into the second phase which performs the actual genetic search.

In the second phase, *report work* will insert the new parameters and their fitness into the population but only if they are better than the worst current member and remove the worst member if required to keep the population size the same. Otherwise the parameters and the result are discarded. Processing a *request work* message will either return a mutation or reproduction (combination) from the population. Section 4.1 describes different methods for this in detail.

This algorithm has significant benefits in heterogeneous environments because the calculation of fitness can be done by each worker concurrently and independently of each other. The algorithm progresses as fast as work is received, and faster workers can processes multiple *request work* messages without waiting on slow workers. This approach is also highly scalable, as the only limiting factor is how fast results can be inserted into the population and how fast *request work* messages can be processed. It is also possible to have multiple masters using an island approach for even greater scalability. This approach is also highly resilient to client side faults, because unreturned work does not effect the server side genetic search.

4.1 Combination Operators

The previous section gave a generic outline for asynchronous search which allows for various combination and mutation operators to be used in generating new parameter sets to be evaluated. This makes the algorithm easily adaptable, which is a desirable quality because no particular search method is ever optimal for all optimization problems. This section describes using the asynchronous search strategy to implement both genetic search (AGS) and particle swarm optimization (APSO).

4.1.1 Asynchronous Genetic Search (AGS)

Asynchronous genetic search generates new individuals using either mutation or a combination operator, as follows.

Mutation Operators

The standard mutation operator for an optimization problem with a continuous domain is to take an individual from the population, and randomly regenerate one of its parameters within the bounds of that parameters. This is often modified so that the range in which the parameter can be modified is gradually reduced as the search progresses, which can result in improved convergence rates.

Average Operator

The standard and most simple combination operator for real variables over a continuous domain is to take the average of two parent individuals and use that as the child.

Double Shot Operator

Desell et al. [12] show that using a double shot operator as opposed to a standard average operator can significantly improve convergence rates for the astronomical modeling application. The double shot operator produces three children instead of one. The first is the average of the two parents, and the other two are located outside the parents, equidistant from the average (see Figure 2). This approach is loosely based on line search, the point outside the more fit parent is in a sense moving down the gradient, while the point outside the less fit parent is moving up the gradient created by the two parents. The motivation for the latter point is to escape local minima.

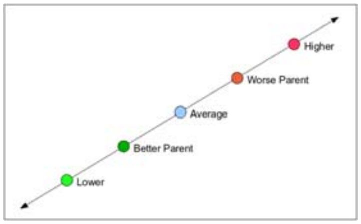


Fig. 2. The double shot operator generates three children: the average, a point outside the worse parent (higher), and a point outside the better parent (lower), the latter two points are a distance from the average equal to the distance between their parents.

Probabilistic Simplex Operator

Unfortunately, all of these approaches require synchrony by creating dependence between fitness calculations. While it is not possible to effectively perform the traditional Nelder-Mead simplex search in a highly heterogeneous

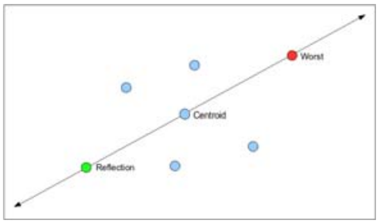


Fig. 3. The simplex method takes the worst point and reflects it through the centroid of the remaining points. The probabilistic simplex operator randomly generates a point on some section of the line connecting the worst point and its reflection.

and volatile environment like BOINC, a probabilistic operator can mimic its behavior. The Nelder-Mead simplex search takes $N + 1$ sets of parameters, and performs *reflection*, *contraction* and *expansion* operators between the worst set of parameters and the centroid of the remaining N (see Figure 3). After calculating the centroid, a line search is performed by expanding or contracting the simplex along this line. Because in our asynchronous model it is not possible to iteratively perform expansions and contractions, a random point is selected on the line joining the worst point and its reflection. There are three parameters involved in this operator, N, the number of points used to form the simplex (chosen randomly from the population), and two limits l_1 and l_2 which specify where on the line the point can be generated. For example, $l_1 = -1$ would set one limit to the reflection and $l_2 = 1$ would set the other limit to the worst point. For the purposes of this study, we use $l_1 = -1.5$ and $l_2 = 1.5$ and examine how children generated from different parts of this line effect the

4.1.2 Asynchronous Particle Swarm Optimization (APSO)

Particle swarm optimization was initially introduced by Kennedy and Eberhart [31, 16] and is a population based global optimization method based on biological swarm intelligence, such as bird flocking, fish schooling, etc. This approach consists of a population of particles, which "fly" through the search space based on their previous velocity, their individual best found position (cognitive intelligence) and the global best found position (social intelligence). The population of particles is updated iteratively as follows, where x is the position of the particle at iteration t, v is it's velocity, p is the individual best for that particle, and g is the global best position. Two user defined constants, c_1 and c_2, allow modification of the balance between local (cognitive) and global (social) search, while another constant the *inertia weight*, w, scales the particle's previous velocity. The search updates the positions and velocities of the particles as follows:

$$v_i(t+1) = w * v_i(t) + c_1 * rand() * (p_i - x_i(t)) + c_2 * rand() * (g_i - x_i(t)) \quad (1)$$

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (2)$$

In a parallel computing scenario, the search typically progresses iteratively. The fitness of each particle is computed in parallel, then the local and global best points are updated. Following this a new positions for each particle are computed and the process repeats.

PSO can be made asynchronous by noting that the method in which the particles move around the search space is not completely dependent on the fitness computed in the previous iteration. A particle will continue to move using its previous velocity and the current local and global best positions found until a new local or global best position is found. By relaxing this restriction slightly by allowing to a particle to continue to move in absence of knowledge of the fitness of previous states, we can utilize the asynchronous search strategy and remove the scalability limitations of traditional PSO.

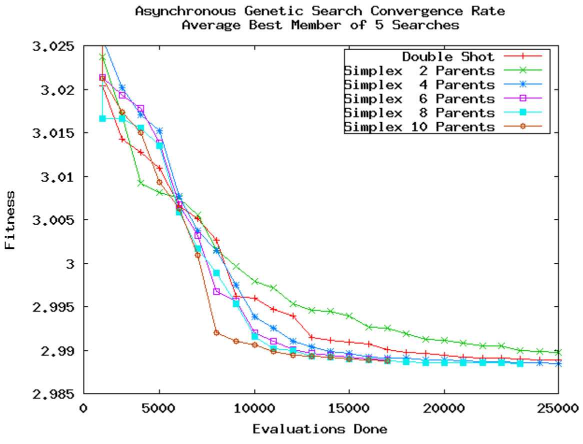
APSO works as follows. New positions for particles are generated in a round robin fashion in response to *request work* messages. Instead of waiting for previously sent out particles to be evaluated, new particles are generated using the current *known global best* and *known local best*. This allows the search to progress and generate new particles asynchronously. When a particle is evaluated, its fitness, position, velocity are reported and the search is updated if the particle is a new local or global best. In this way the APSO performs nearly identically to PSO, without scalability limitations. As more processors request work, particles are generated further ahead increasing the exploratory component of the search.

5 MilkyWay@Home: Finding Galactic Substructure Using Genetic Search on Volunteered Computers

5.1 Convergence

5.1.1 Asynchronous Genetic Search

The hybrid simplex method was evaluated using the astronomical modeling problem detailed by Purnell et al [43]. Performing the evaluation of a single model to a small wedge of the sky consisting of approximately 200,000 stars can take between 15 minutes to an hour on a single high end processor. Because of this, to be able to determine the globally optimal model for that wedge in any tractable amount of time requires extremely high powered computing environments. To measure the effect of asynchronicity on the hybrid genetic search, both synchronous and asynchronous computing environments are used, 1024 processors of an IBM BlueGene/L and a BOINC volunteer computing project with over 1,000 volunteered computers.



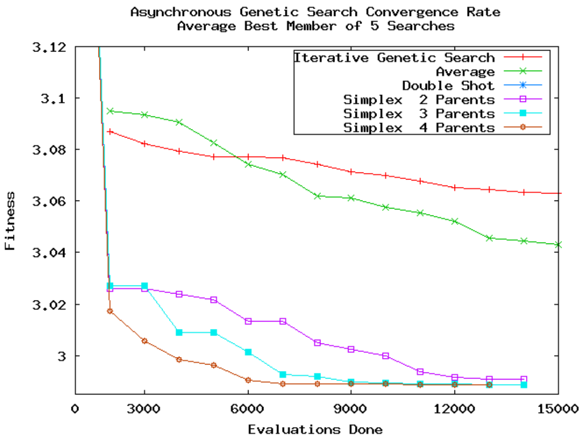


Fig. 4. Fitness of the best member found averaged over five searches for the double shot approach and the simplex hybrid with $N = 2..5$, using the BOINC volunteered computers and the BlueGene supercomputer.

Figure 4 shows the performance of the double shot approach, and the simplex approach with varying numbers of parents N being used to calculate the centroid on both environments. Previous work has shown that the double shot approach significantly outperforms iterative genetic search and asynchronous genetic search using only the average and mutation operators [49]. All approaches converged to the known global optimum of the data. For both computing environments, a population of size 300 was used, and the mutation operator was applied 20% of the time, all other members were generated with the corresponding operator. The range of the probabilistic line search for the simplex hybrid was defined by the limits $l_1 = -1.5$ and $l_2 = 1.5$. For the synchronous execution on the BlueGene, each model evaluation was performed by dividing the work over the 1024 processors, and immediately attempting to insert the member into the population - in this way only the most evolved population was used to generate new members and the population was continuously updated. The asynchronous execution on BOINC generates new members from the current population whenever users request more work. After a user has completed the evaluation of a member, it's sent to the server and inserted into the population. There is no guarantee of when the fitness of a generated member will be returned, or even if it will be returned at all.

On the BlueGene, the hybrid simplex method shows dramatic improvement over the double shot approach, with the difference increasing as more parents are used to calculate the centroid. While the double shot method typically converges in around 18,000 iterations, the simplex hybrid with $N = 4$ converges in approximately 8,000. Compared to the 50,000 iterations reported for traditional iterative genetic search [12], the convergence rate is excellent. Using BOINC shows similar results, however the convergence rates are not as fast on the BlueGene, which is to be expected. Generally, increasing the number of points used to calculate the centroid results in better searches, however on BOINC the simplex with $N = 2$ and double shot operators initially seem to converge more quickly than the more informed simplex with $N = 4..10$, which was not the case on the BlueGene. The asynchronous approach on BOINC may take more iterations, but BOINC is much more accessible as it is dedicated to the project at hand, while use of the BlueGene is shared among many researchers. Because of this, even though the quantity of fitness evaluations done per second is similar for both computing environments, the BOINC framework can perform more searches and does so at a fraction of the cost. These results are very promising for the use of asynchronous search and volunteer computing for computationally intensive scientific modeling.

5.1.2 Asynchronous Particle Swarm Optimization

Asynchronous particle swarm optimization (APSO) was also tested using the BOINC computing platform, with results competitive to asynchronous genetic search. Figure 5 shows how the search progressed using different values of the inertia weight w. Typically, regular particle swarm optimization uses $w = 1$, $c1 = 2$, and $c2 = 2$ as standard values for constants, however we found that in an asynchronous setting, lower values of w performed significantly better. While all values did eventually reach the global optimum, a value of 0.4 tended to find the optimum the fastest. It is however interesting to note that while values 0.6 and 0.8 initially converge faster, they then spend much longer zeroing in on the correct value. It could be said that this is evidence of a higher inertia weight being more exploratory, finding good possible areas to search, but these higher values lack the required exploitation of those good areas needed to ultimately find a correct value. It also supports results found by Dingxue et al. which find that using an adaptive value for w improves convergence [13].

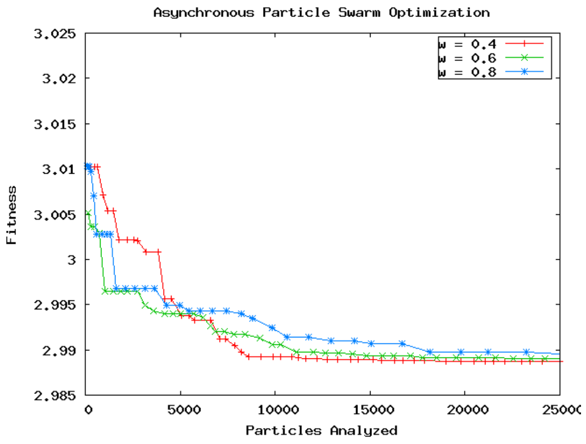


Fig. 5. Fitness of the best particle found averaged over five searches for asynchronous particle swarm optimization on the BOINC volunteered computers, using constants $c1 = c2 = 2.0$ and inertia weight $w = 0.4$, 0.6 and 0.8.

5.2 Operator Analysis

To better understand the effect of the operators in evolving the population, as well as the effect of asynchronicity and of a highly heterogeneous computing environment on the fitness returned, the number of members processed between the generation and reporting of a members fitness was tracked, as well as information about how it was generated. For both environments, the best N was used. Figure 6 shows the percentage of members inserted into the population and at what position in the population they were inserted based on what part of the line they were generated with using the simplex hybrid with $N = 4$ on the BlueGene. The population is sorted from the best fit to the least, so the lower the position at which a member is inserted, the better its fitness with respect to the rest of the population. Figures 7 and 8 show the same information for BOINC and $N = 4$. To provide a measure of how far the population evolved while a member was being evaluated, these results are partitioned by how many other members were reported before the fitness of the current member was reported. The range of the probabilistic line search for the simplex was defined by limits $l_1 = -1.5$ and $l_2 = 1.5$ and the statistics are taken from five separate searches.

On the BlueGene, the best insert rate and quality was from points around the centroid (generated between limits of 0.5 and -0.5). While inside of the worst point (1.0 to 0.5) had the highest insert rate, the quality of inserted members was rather low. Points near the reflection of the worst point through the centroid (-1.5 to -0.5) tended to have low insert rates, however when they were inserted they tended to be very fit. Points outside of the worst

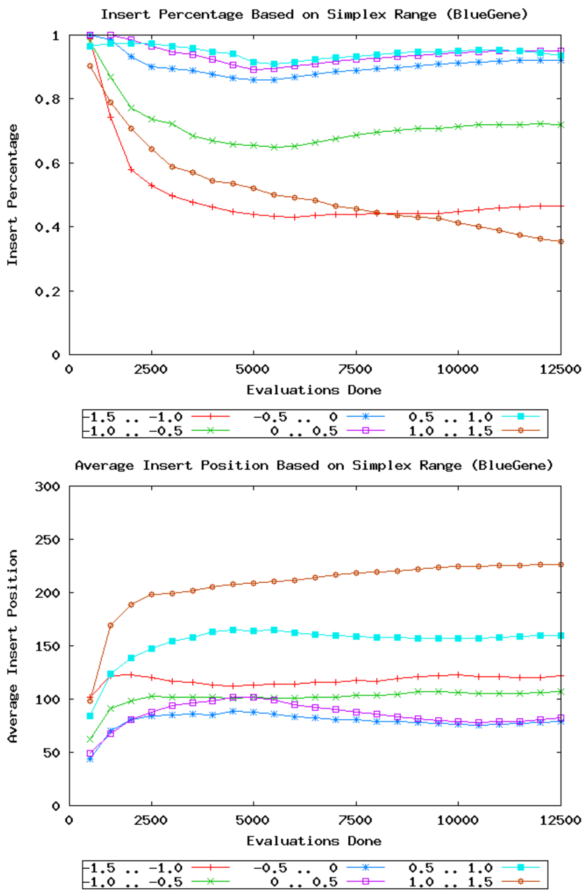


Fig. 6. Average insert rate and insert position of members based on what part of the line calculated by the simplex hybrid they were generated on, for $N = 4$ using the BlueGene supercomputer. A lower insert position means the member is more fit than the rest of the population.

member (1.0 to 1.5) had the worst insert rate and the least fit. These results suggest that the probabilistic simplex search could be further optimized by restricting the range to limits $l_1 = -1.5$ and $l_2 = 0.5$, by eliminating the poorest performing range of 0.5 to 1.5.

BOINC showed similar results for quickly reported results (less than 200 members reported while the member was being evaluated) with points generated near the centroid (-0.5 to 0.5) having the best fitness and insert rate (see Figures 7 and 8). One notable exception was that points generated on the inside of the worst point (0.5 to 1.0) had a notably lower insert rate and that points generated near the worst point (0.5 to 1.5) quickly degraded in

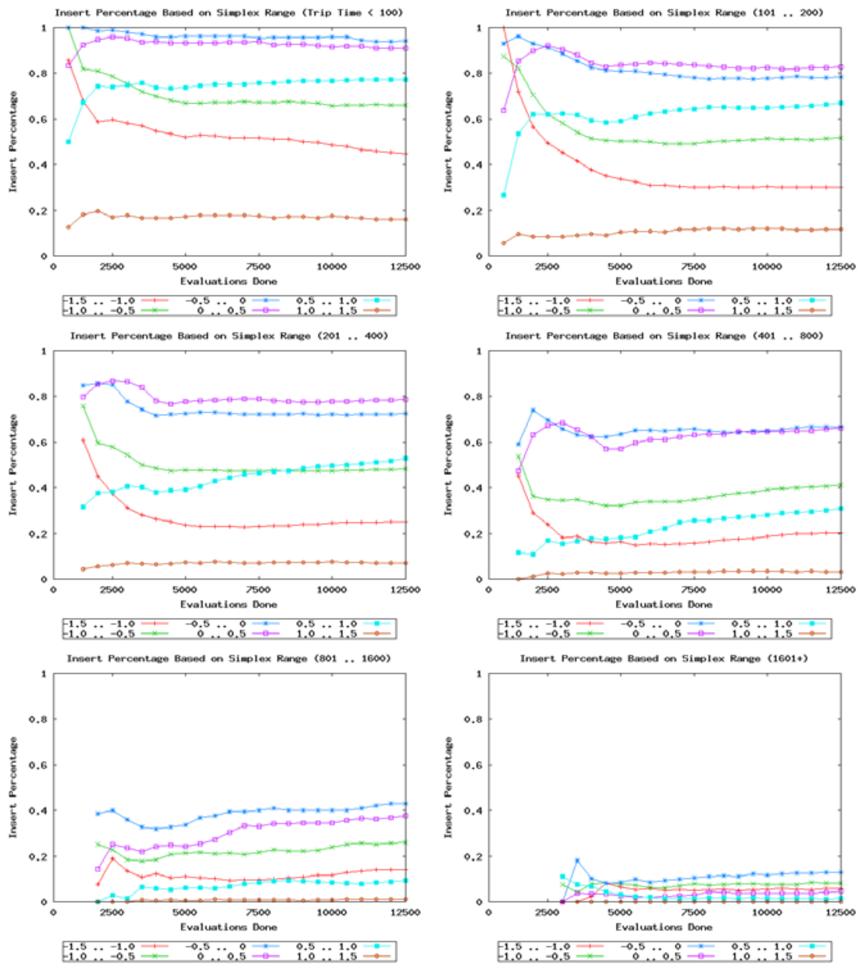


Fig. 7. Average insert rate of members based on what part of the line calculated by the simplex hybrid they were generated on, for $N = 5$ using the BOINC framework. The results are partitioned by how many other members were reported while the used members were being generated (0..100 to 1601+) to show the effects of asynchronicity and of a heterogeneous computing environment.

terms of insert rate compared to other points. With over 1600 evaluations being reported during a members round trip time, not a single point generated past the worst point was inserted. Another point of interest is that while points generated near the reflection (-1.5 to -0.5) had lower insertion rates than those near the centroid (-0.5 to 0.5), as the report time increased, their average insert position stayed the same and eventually had better fitness than points generated near the centroid. As with the BlueGene, the results suggest

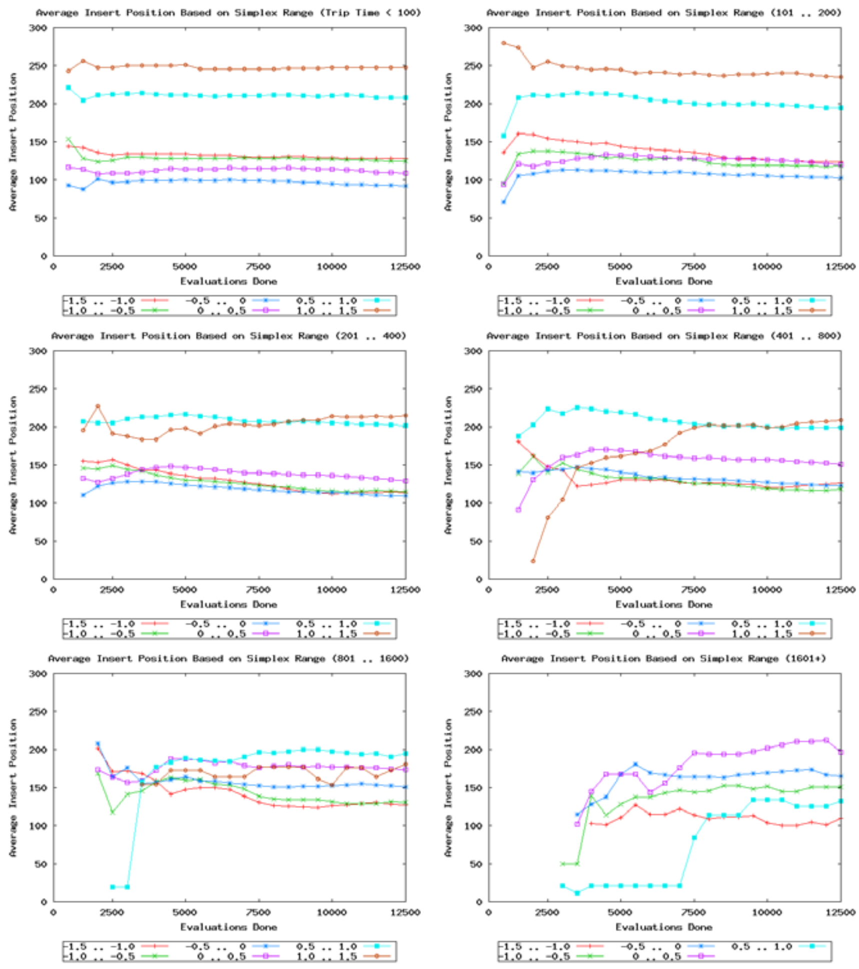


Fig. 8. Average insert position of members based on what part of the line calculated by the simplex hybrid they were generated on, for $N = 4$ using the BOINC framework. A lower insert position means the member is more fit than the rest of the population. The results are partitioned by how many other members were reported while the used members were being generated (0..100 to 1601+) to show the effects of asynchronicity and of a heterogeneous computing environment.

that refining the limit on the probabilistic simplex operator to $l_1 = -1.5$ and $l_2 = 0.5$ would improve the convergence rates. Additionally, it appears that the result report time does have an effect on which part of the line used by the probabilistic simplex operator is better to draw new members from. An intelligent work scheduling mechanism could assign members generated near the reflection to processors with slower reporting times, and those generated near the centroid to processors with faster reporting times. Also, as the search progresses, there are fluctuations as to where the best points are generated from. An adaptive search could refine the limits to improve convergence rates. It is important to note that even the slowest processors retain their ability to evaluate members that are of benefit to the search, which is an important attribute for any algorithm running on massively distributed and heterogeneous environments.

6 Conclusions

Volunteer computing platforms can significantly enable scientists using evolutionary algorithms by providing them access to thousands of processors worldwide. The heterogeneity inherent in this worldwide computing infrastructure can be tackled by using asynchronous versions of evolutionary algorithms, which are better suited to deal with the wide variety of processing speeds and failure characteristics found in volunteer computing environments.

We have shown a specific application in astronomy, MilkyWay@Home, that uses asynchronous genetic search on BOINC to discover substructure in the Milky Way Galaxy from Sloan Digital Sky Survey data. The availability of more processing power for scientists has the potential to enable better science: more complex models can be tested on larger data sets, streamlining the scientific process.

Additional research directions include adapting additional evolutionary algorithms to the heterogeneous and failure-prone nature of volunteer computing environments, creating generalized scientific computing frameworks that lower the barrier of entry to new scientific domains, and developing hybrid mechanisms that can make efficient use of diverse distributed computing environments including supercomputers, grids, clusters, clouds and the Internet.

Acknowledgments

The U.S.-based authors were partially supported by the National Science Foundation under Grants No. 0612213, 0607618 and 0448407. The Spain-based authors were partially supported by Regional Gridex project PRI06A223 Junta de Extremadura and National Nohnes project TIN2007-68083-C02-01 Spanish Ministry of Science and Education.

References

1. Allen, M.: Do it yourself climate prediction. Nature (1999)
2. Anderson, D.: Boinc: a system for public-resource computing and storage. In: Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4–10 (2004)

3. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Seti@home: an experiment in public-resource computing. Commun. ACM 45(11), 56–61 (2002), http://doi.acm.org/10.1145/581571.581573
4. Anderson, D.P., Fedak, G.: The computational and storage potential of volunteer computing. In: Proceedings of CCGRID, pp. 73–80 (2006)
5. Andre, D., Koza, J.R.: Parallel genetic programming: a scalable implementation using the transputer network architecture, pp. 317–337 (1996)
6. Belokurov, V., Zucker, D.B., Evans, N.W., Gilmore, G., Vidrih, S., Bramich, D.M., Newberg, H.J., Wyse, R.F.G., Irwin, M.J., Fellhauer, M., Hewett, P.C., Walton, N.A., Wilkinson, M.I., Cole, N., Yanny, B., Rockosi, C.M., Beers, T.C., Bell, E.F., Brinkmann, J., Ivezić, Ž., Lupton, R.: The Field of Streams: Sagittarius and Its Siblings. Astrophysical Journal Letters 642, L137–L140 (2006)
7. Bennet, F.H.I., Koza, J.R., Shipman, J., Stiffelman, O.: Building a parallel computer system for \$18,000 that performs a half peta-flop per day. In: Proceedings of the Genetic and Evolutionary Computation Conference, Orlando, Florida, USA, pp. 1484–1490 (1999)
8. Cahon, S., Melab, N., Talbi, E.: ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. Journal of Heuristics 10(3), 357–380 (2004)
9. Cantu-Paz, E.: A survey of parallel genetic algorithms. Calculateurs Paralleles, Reseaux et Systems Repartis 10(2), 141–171 (1998)
10. Cole, N., Jo Newberg, H., Magdon-Ismail, M., Desell, T., Szymanski, B., Varela, C.: Tracing the Sagittarius Tidal Stream with Maximum Likelihood. In: American Institute of Physics Conference Series, vol. 1082, pp. 216–220 (2008), doi:10.1063/1.3059049
11. Cole, N., Newberg, H.J., Magdon-Ismail, M., Desell, T., Dawsey, K., Hayashi, W., Liu, X.F., Purnell, J., Szymanski, B., Varela, C., Willett, B., Wisniewski, J.: Maximum Likelihood Fitting of Tidal Streams with Application to the Sagittarius Dwarf Tidal Tails. The Astrophysical Journal 683, 750–766 (2008)
12. Desell, T., Cole, N., Magdon-Ismail, M., Newberg, H., Szymanski, B., Varela, C.: Distributed and generic maximum likelihood evaluation. In: 3rd IEEE International Conference on e-Science and Grid Computing (eScience2007), Bangalore, India, pp. 337–344 (2007)
13. Dingxue, Z., Zhihong, G., Xinzhi, L.: An adaptive particle swarm optimization algorithm and simulation. In: IEEE International Conference on Automation and Logistics, pp. 2399–2042 (2007)
14. Bill Punch, D.Z.: Lil-gp, http://garage.cse.msu.edu/software/lil-gp/index.html
15. Duffau, S., Zinn, R., Vivas, A.K., Carraro, G., Méndez, R.A., Winnick, R., Gallart, C.: Spectroscopy of QUEST RR Lyrae Variables: The New Virgo Stellar Stream. The Astrophysical Journal Letters 636, L97–L100 (2006)
16. Eberhart, R.C., Kennedy, J.: A new optimizer using particle swarm theory. In: Sixth International Symposium on Micromachine and Human Science, pp. 33–43 (1995)
17. Eggen, O.J., Lynden-Bell, D., Sandage, A.R.: Evidence from the motions of old stars that the Galaxy collapsed. The Astrophysical Journal 136, 748 (1962)
18. Fernández, F., Tomassini, M.L.V.: An empirical study of multipopulation genetic programming. Genetic Programming and Evolvable Machines (2003)

19. Fedak, G., Germain, C., Neri, V., Cappello, F.: XtremWeb: A Generic Global Computing System. In: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2001) (2001)
20. Fernandez, F., Spezzano, G., Tomassini, M., Vanneschi, L.: Parallel genetic programming. In: Alba, E. (ed.) Parallel Metaheuristics, Parallel and Distributed Computing, ch. 6, pp. 127–153. Wiley-Interscience, Hoboken (2005)
21. Fernández, F., Cantú-Paz, E.: Special issue parallel bioinspired algorithms. Journal of Parallel and Distributed Computing 66(8) (2006)
22. Foster, I., Kesselman, C.: Globus: A metacomputing infrastructure toolkit. The International Journal of Supercomputer Applications and High Performance Computing 11(2), 115–128 (1997), citeseer.ist.psu.edu/article/foster96globus.html
23. Foundation, F.S.: Gnu lesser general public license, version 3, http://www.gnu.org/licenses/lgpl-3.0.html
24. Freeman, K., Bland-Hawthorn, J.: The New Galaxy: Signatures of Its Formation. Annual Review of Astronomy & Astrophysics 40, 487–537 (2002)
25. Freeman, K.C.: The Galactic spheroid and old disk. Annual Review of Astronomy & Astrophysics 25, 603–632 (1987)
26. Gagné, C., Parizeau, M., Dubreuil, M.: Distributed beagle: An environment for parallel and distributed evolutionary computations. In: Proc. of the 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS) 2003, pp. 201–208 (2003)
27. González, D.L., de Vega, F.F., Trujillo, L., Olague, G., Araujo, L., Castillo, P., Merelo, J.J., Sharman, K.: Increasing gp computing power for free via desktop grid computing and virtualization. In: Proceedings of the 17th Euromicro Conference on Parallel, Distributed and Network-based Processing, Weimar, Germany, pp. 419–423 (2009)
28. Ibata, R., Irwin, M., Lewis, G.F., Stolte, A.: Galactic Halo Substructure in the Sloan Digital Sky Survey: The Ancient Tidal Stream from the Sagittarius Dwarf Galaxy. The Astrophysical Journal Letters 547, L133–L136 (2001)
29. Ibata, R., Lewis, G.F., Irwin, M., Totten, E., Quinn, T.: Great Circle Tidal Streams: Evidence for a Nearly Spherical Massive Dark Halo around the Milky Way. The Astrophysical Journal 551, 294–311 (2001)
30. Ibata, R.A., Gilmore, G., Irwin, M.J.: A Dwarf Satellite Galaxy in Sagittarius. Nature 370, 194 (1994)
31. Kennedy, J., Eberhart, R.C.: Particle swarm optimization. In: IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948 (1995)
32. Kesselman, C., Foster, I.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
33. Laure, E., Fisher, S., Frohner, A., Grandi, C., Kunszt, P., Krenek, A., Mulmo, O., Pacini, F., Prelz, F., White, J., et al.: Programming the Grid with gLite. Computational Methods in Science and Technology 12(1), 33–45 (2006)
34. Law, D.R., Johnston, K.V., Majewski, S.R.: A Two Micron All-Sky Survey View of the Sagittarius Dwarf Galaxy. IV. Modeling the Sagittarius Tidal Tails. The Astrophysical Journal 619, 807–823 (2005)
35. Lombraña, D., Fernández, F., Trujillo, L., Olague, G., Cárdenas, M., Araujo, L., Castillo, P., Sharman, K., Silva, A.: Interpreted applications within boinc infrastructure. In: Ibergrid 2008, Porto, Portugal, pp. 261–272 (2008)
36. Luke, S., Panait, L., Balan, G., Paus, S., Skolicki, Z., Popovici, E., Harrison, J., Bassett, J., Hubley, R., Chircop, A.: Ecj a java-based evolutionary computation research system (2007), http://cs.gmu.edu/~eclab/projects/ecj/

37. Litzkow, M., Tannenbaum, T., Basney, J., Livny, M.: Checkpoint and migration of unix processes in the condor distributed processing system. Tech. rep., University of Wisconsin (1997)
38. McNett, D.: Rc5-64 has been solved! `http://www.distributed.net/pressroom/news-20020926.txt`
39. McNett, D.: Secure encryption challenged by internet-linked computers, `http://www.distributed.net/pressroom/56-PR.html`
40. Newberg, H.J., Yanny, B., Cole, N., Beers, T.C., Re Fiorentin, P., Schneider, D.P., Wilhelm, R.: The Overdensity in Virgo, Sagittarius Debris, and the Asymmetric Spheroid. The Astrophysical Journal 668, 221–235 (2007)
41. Newberg, H.J., Yanny, B., Rockosi, C., Grebel, E.K., Rix, H.W., Brinkmann, J., Csabai, I., Hennessy, G., Hindsley, R.B., Ibata, R., Ivezić, Z., Lamb, D., Nash, E.T., Odenkirchen, M., Rave, H.A., Schneider, D.P., Smith, J.A., Stolte, A., York, D.G.: The Ghost of Sagittarius and Lumps in the Halo of the Milky Way. The Astrophysical Journal 569, 245–274 (2002)
42. de la, O, F.C., Guisado, J.L., Lombraña, D., Fernández, F.: Una herramienta de programación genética paralela que aprovecha recursos públicos de computación. In: V Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, Tenerife, Spain, pp. 167–173 (2007)
43. Purnell, J., Magdon-Ismail, M., Newberg, H.: A probabilistic approach to finding geometric objects in spatial datasets of the Milky Way. In: Hacid, M.-S., Murray, N.V., Raś, Z.W., Tsumoto, S. (eds.) ISMIS 2005. LNCS (LNAI), vol. 3488, pp. 475–484. Springer, Heidelberg (2005)
44. Pande Vijay, S., Ian, B., Jarrod, C., Elmer Sidney, P., Siraj, K., Larson Stefan, M., Young Min, R., Shirts Michael, R., Snow Christopher, D., Sorin Eric, J., Zagrovic, B.: Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. Biopolymers 68, 91–109 (2003)
45. Samples, M., Daida, J., Byom, M., Pizzimenti, M.: Parameter sweeps for exploring GP parameters. In: Proceedings of the 2005 workshops on Genetic and evolutionary computation, pp. 212–219 (2005)
46. Searle, L., Zinn, R.: Compositions of halo clusters and the formation of the galactic halo. The Astrophysical Journal 225, 357–379 (1978)
47. Sintes, A.M.: Recent results on the search for continuous sources with ligo and geo600 (2005), arXiv.org
48. Storn, R., Price, K.: Minimizing the real functions of the icec 1996 contest by differential evolution. In: Proceedings of the IEEE International Conference on Evolutionary Computation, Nagoya, Japan, pp. 842–844 (1996)
49. Szymanski, B., Desell, T., Varela, C.: The effect of heterogeneity on asynchronous panmictic genetic search. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 457–468. Springer, Heidelberg (2008)
50. Team, D.: distributed.net, `http://www.distributed.net`
51. Team, D.: Optimal golomb ruler, `http://www.distributed.net/ogr/`
52. Team, G.: Greate internet mersenne prime search, `http://www.mersenne.org`
53. Tomassini, M.: Spatially Structured Evolutionary Algorithms. Springer, Heidelberg (2005)
54. Trujillo, L., Olague, G.: Automated Design of Image Operators that Detect Interest Points, pp. 483–507. MIT Press, Cambridge (2008)

55. Vivas, A.K., Zinn, R., Andrews, P., Bailyn, C., Baltay, C., Coppi, P., Ellman, N., Girard, T., Rabinowitz, D., Schaefer, B., Shin, J., Snyder, J., Sofia, S., van Altena, W., Abad, C., Bongiovanni, A., Briceño, C., Bruzual, G., Della Prugna, F., Herrera, D., Magris, G., Mateu, J., Pacheco, R., Sánchez, G., Sánchez, G., Schenner, H., Stock, J., Vicente, B., Vieira, K., Ferrín, I., Hernandez, J., Gebhard, M., Honeycutt, R., Mufson, S., Musser, J., Rengstorf, A.: The QUEST RR Lyrae Survey: Confirmation of the Clump at 50 Kiloparsecs and Other Overdensities in the Outer Halo. The Astrophysical Journal Letters 554, L33–L36 (2001)
56. Willman, B., Dalcanton, J.J., Martinez-Delgado, D., West, A.A., Blanton, M.R., Hogg, D.W., Barentine, J.C., Brewington, H.J., Harvanek, M., Kleinman, S.J., Krzesinski, J., Long, D., Neilsen Jr., E.H., Nitta, A., Snedden, S.A.: A New Milky Way Dwarf Galaxy in Ursa Major. The Astrophysical Journal Letters 626, L85–L88 (2005)
57. Yanny, B., Newberg, H.J., Grebel, E.K., Kent, S., Odenkirchen, M., Rockosi, C.M., Schlegel, D., Subbarao, M., Brinkmann, J., Fukugita, M., Ivezic, Ž., Lamb, D.Q., Schneider, D.P., York, D.G.: A Low-Latitude Halo Stream around the Milky Way. The Astrophysical Journal 588, 824–841 (2003)
58. Yanny, B., Newberg, H.J., Kent, S., Laurent-Muehleisen, S.A., Pier, J.R., Richards, G.T., Stoughton, C., Anderson Jr., J.E., Annis, J., Brinkmann, J., Chen, B., Csabai, I., Doi, M., Fukugita, M., Hennessy, G.S., Ivezić, Ž., Knapp, G.R., Lupton, R., Munn, J.A., Nash, T., Rockosi, C.M., Schneider, D.P., Smith, J.A., York, D.G.: Identification of A-colored Stars and Structure in the Halo of the Milky Way from Sloan Digital Sky Survey Commissioning Data. The Astrophysical Journal 540, 825–841 (2000)
59. Zucker, D.B., Belokurov, V., Evans, N.W., Wilkinson, M.I., Irwin, M.J., Sivarani, T., Hodgkin, S., Bramich, D.M., Irwin, J.M., Gilmore, G., Willman, B., Vidrih, S., Fellhauer, M., Hewett, P.C., Beers, T.C., Bell, E.F., Grebel, E.K., Schneider, D.P., Newberg, H.J., Wyse, R.F.G., Rockosi, C.M., Yanny, B., Lupton, R., Smith, J.A., Barentine, J.C., Brewington, H., Brinkmann, J., Harvanek, M., Kleinman, S.J., Krzesinski, J., Long, D., Nitta, A., Snedden, S.A.: A New Milky Way Dwarf Satellite in Canes Venatici. The Astrophysical Journal Letters 643, L103–L106 (2006)

Self-coordinated On-Chip Parallel Computing: A Swarm Intelligence Approach

Danilo Pani and Luigi Raffo

Abstract. Self organization is the property of some natural systems to organize themselves without a central coordination unit to perform specific tasks. Swarm Intelligence is a bioinspired paradigm coming from the observation of natural swarms, such as honey bees and bird flocks. Swarms exploit self organization to achieve coordination, speed-up and fault tolerance. This interesting paradigm has been applied in different research fields, mainly in robotics and optimization algorithms. Our pioneering studies about the application of this powerful paradigm to digital VLSI systems demonstrated that Swarm Intelligence can be applied to the design of scalable computing architectures composed of a large set of self-coordinated hardware agents. In this Chapter we present this approach with a review of our research works in this field from the first explorations to the latest results: the FPGA implementation of a coprocessing architecture expressly conceived resorting to the Swarm Intelligence principles. Some experimental results are presented to evaluate the main features of this innovative approach, which shows interesting performance improvements without any programming effort and without complex tools for compilation and mapping, compared to other state-of-the-art coprocessing architectures.

1 Introduction

Multimedia, cryptography, scientific computations and many current applications require growing computational capabilities. Interactive applications are characterized by fluctuations in the workload at run time that depend on the user's choices and are unpredictable at compilation time. A great flexibility in multiple tasks execution to adapt the system responses to the different loads is then required. The development of efficient digital architectures, able to support this kind of intensive

Danilo Pani · Luigi Raffo
DIEE - University of Cagliari, Piazza d'Armi, Cagliari, Italy
e-mail: {pani,luigi}@diee.unica.it

computations even in multitasking scenarios, represents one of the primary goals of today's digital designers.

To achieve this goal, the traditional Von Neumann and Harvard architectural models, implicitly based on the assumption that the space on chip is a limited resource so that it is better to trade "space for time" [18], need to be overcome introducing explicit parallelism exploitation. It has been demonstrated that traditional processing cores of today's size do not scale well to future technologies [5]. As a matter of fact, CMOS scaling is leading wire delay and power to be the most important performance limiters. Rethinking the conventional approach to microprocessor design, focusing on scalable and distributed alternatives to current centralized microprocessors, is necessary to face the new technological challenges [29]. Furthermore, in deep sub-micron technologies, high clock rates and densities will make chips more prone to permanent and transient faults at run time [31], whereas production processes are less reliable with the reduction of the transistors size, causing an increase in defects at production time. Large monolithic cores are completely unable to face this new challenge.

One of the main problems of single-chip multicore solutions, and probably the biggest one, is programmability since for several years manufacturers have retained the uniprocessor monolithic design approach to provide software compatibility [14]. Even if the multicore model suggests a great flexibility, the commercial multicore architectures (with more than 256 cores on the same chip) strongly relies on design-time resources mapping to deal with the huge hardware complexity [1, 3], thus limiting flexibility and fault-tolerance, while leaving to the programmer the role to explicitly exploit parallelism. This is also true, with several relaxed constraints, for less application-specific platforms characterized by a smaller number of cores [4].

In this scenario, it seems clear that some new ideas are required to evolve from the "centralized mindset" towards decentralization and self organization [30], to support the development of scalable massively parallel architectures able to provide an easier control while guaranteeing high performance, flexibility, multitasking and fault tolerance. Some natural systems exhibit these properties, employed for different purposes. Swarm Intelligence (SI) [20] is a bioinspired paradigm that takes inspiration from natural swarms, large sets of simple individuals with limited capabilities able to carry out complex tasks exploiting cooperation and self organization. This interesting paradigm has been applied in many research fields, mainly in robotics and optimization algorithms [9]. Recently we demonstrated the possibility to use it for the design of scalable computational tiled architectures composed of a large set of self-coordinated "hardware agents" [27, 6, 10].

This chapter presents the latest advances in SI applied to this field. SI, and explicitly the cooperative behaviors behind the self-coordination in swarm systems, was used for different purposes: self organization of the processing without low-level centralized control and without complex software toolchains, automatic load balancing, simultaneous multitasking and fault tolerance. A brief scientific review of the previous works aimed to exploit SI for the design of digital computational platforms is presented, starting from the absolutely first studies in the field to arrive to the latest developments, a coprocessing unit for fixed point array processing.

The effectiveness and the limits of the proposed approach, which shows significant performance improvements without any programming effort and without complex tools for compilation and mapping, are discussed. The architecture also supports fault tolerance by cell exclusion, effective in reducing the influence of the presence of faulty elements without spare resources. Since the fault tolerance characteristics are quite complex, they are not presented in details in this Chapter.

2 Background on Swarm Intelligence

It is quite difficult to give a precise definition of Swarm Intelligence. Several definitions already exist and somehow point out different aspects of the same idea. The term was originally coined by Beni in 1988 [8] in the context of cellular robotic systems where many simple agents occupy one- or two-dimensional environments to generate patterns and self-organize through nearest-neighbor interactions. To explain the term *swarm*, Kennedy et al. [20] quote a FAQ document from Santa Fe Institute about the Swarm simulation system telling that:

> We use the term “swarm” in a general sense to refer to any such loosely structured collection of interacting agents. The classic example of a swarm is a swarm of bees, but the metaphor of a swarm can be extended to other systems with a similar architecture. An ant colony can be thought of as a swarm whose individual agents are ants, [...] an immune system is a swarm of cells and molecules, and an economy is a swarm of economic agents. Although the notion of a swarm suggests an aspect of collective motion in space, as in the swarm of a flock of birds, we are interested in all types of collective behavior, not just spatial motion.

In this sense SI is simply the emergent collective intelligence of groups of simple autonomous agents with global adaptive behavior [9]. The term “agent” should be carefully considered since it may generate confusion. Here, an autonomous agent is a subsystem that interacts with its environment, which probably consists of other agents, but acts relatively independently from all other agents. The autonomous agent does not follow commands from a leader [23].

Natural swarms largely rely on self organization, resorting to simple local interactions to coordinate the activity of the whole colony. Since the whole system is not a single entity but rather a self-coordinated set of elementary entities, it is able to plastically adapt itself to different environmental conditions without any centralized control. Self organization is responsible for agents recruitment and tasks execution. Cooperation inside the colony allows to adapt the effort in a task execution to the available resources, that is very important when more than one task needs to be executed at the same time and tasks arise in run-time unpredictably. The loose structure of swarms entails several consequences very useful for digital architectures design. For instance, a natural swarm is intrinsically fault tolerant, and performance in a task execution can be modulated by the number of agents involved, until a reasonable threshold is reached (performance scalability).

Communication within swarms is accomplished in different ways but it is always true that the information exchange is local and the information is also local, i.e. each agent does not have the perception of the whole system, but only of parts of it. Local interactions can be either direct (such as antennation, visual contact, etc.) or indirect (mediated by the environment). In the first case, interactions require the simultaneous involvement of two or more individuals, whereas in the second one they could take place asynchronously with some delay. *Stigmergy* [17] is a form of indirect interaction mediated by the environment used by some natural swarms to implicitly coordinate their activities. This strategy allows environmental modifications and agent responses to take place in different times [9]. An example of stigmergy is the clustering behavior typical of some species of ants, leading to cemetery organization, larval sorting, etc. Those ants move randomly in the nest, picking up the items (corpses, larvae, etc.) and putting them down next to other items. Doing this, carriers attach to the items a small quantity of pheromone and, in turn, carriers perceive the presence of clusters of items following pheromone concentrations. Then, clusters become more and more attractive to put down items, so that the overall effect is the formation of large clusters of items in the nest. Since ants do not use direct interactions to accomplish the task, but they perform it perceiving the environmental modifications, this is a natural example of stigmergy.

The powerfulness of stigmergy is that this mechanism implicitly allows large scalability in the size of the group of agents. In fact, one problem is that as the size of the group increases, the number of potential communications increases exponentially and the amount of information being transferred becomes soon unwieldy. With stigmergy, the communications overhead for each agent does not increase with the size of the group: the agent merely follows the same fixed set of rules and modifies its behavior according to the environmental perception. Several other advantages are fault tolerance, adaptation, speed, modularity, autonomy and parallelism [19].

Nowadays SI is exploited in several fields. One pioneering application was in the field of optimization algorithms for hard NP problems [13, 16, 35], with some recent interesting extensions to continuous optimization problems [34]. The particular field of swarm robotic led to the creation of simple little robots realizing real world operative tasks [21, 24, 32, 36]. Adaptive routing algorithms for communication networks were developed starting from SI approaches [33, 11], with several works in the field of wireless sensor networks [12], and even some algorithms were developed for Computer Aided Design (CAD) tools for VLSI design partitioning [22]. In the following of this Chapter we will show the application of SI to the design of innovative parallel distributed system-on-chip architectures.

3 Swarm Architectures: The Proposed Metaphor

Traditional approaches to parallelization are based on a fixed partitioning of the work between the available units, with severe drawbacks: the system shows poor flexibility and is unable to really adapt to different workloads at run-time, although supporting them. The scalability is somehow limited by centralized-control strategies, and fault tolerance is seldom supported so that even in the so called "tiled architectures" [7], the fault of a single unit entails the fault of the whole system. We have seen that natural swarms are able to carry out highly complex tasks exhibiting all the characteristics required to address these issues. The central idea expressed in this Chapter is that it is possible to create a set of interacting elementary hardware agents [26] asking them to carry out efficiently a segment of the current computation but leaving the possibility to *organize themselves* in the best way to perform it. A system can be considered self-organized if it is organized without any external or central dedicated control entity. Obviously there will always be some constraints to be met, so that the system should behave as the user expects, and the user should remain in control at a high level even if without a direct control [15].

3.1 First Explorations

To the best of our knowledge, the first studies aimed to apply SI paradigm to the design of innovative digital architectures date 2004 [25, 26]. In those studies, we introduced cooperative processing with low-level hardware agents to implement a common Digital Signal Processing task. In this case, hardware agents were simple sequential multipliers with an elementary smart control unit. A swarm of 8 of these agents was able to cooperate by means of simple rules to perform the sum of 8 products. The system adopted an improvement of the sequential Modified Booth Algorithm (MBA), which analyzes the multiplier operand split into triplets with one bit overlap [28] and exploits cooperation at sub-operation level (triplets processing) between the agents without any centralized control.

In this case, we chose the direct interactions on a shared bus (bus arbitrage was managed in a distributed way) for agent communications. This choice reduces concurrency, thus limiting the cooperation. Nevertheless, the swarm approach leads to an amplification effect in the execution speed, i.e. by means of cooperation the system achieves better performance compared to the case of simple parallel processing with fixed partitioning on the same architecture or equivalents. This is due to the fact that agents exploit the reformulated MBA and then perform a reduced number of operation steps as a function of data complexity, which is related to the triplets content [26]. This implies that starting from 8 multiplications, not all of them will require the same processing time, and then cooperation can take place allowing unloaded agents to help overloaded ones. The result is an overall latency reduction, as shown in Figure 1.

The main limit of the system, developed as a proof-of-concept architecture, is the reduced application field. Limitedly to the exploitation of SI, it presents other three main drawbacks: it is potentially able to support fault tolerance but was not designed for that purpose, architectural scaling is limited by the bus-based structure even if some strategies have been proposed [26], and concurrency is limited by both direct interaction scheme (requiring a synchronization) and again bus-based structure. Furthermore, the hardware overhead required to produce the computational substrate was too high with respect to the required functionality, suggesting an

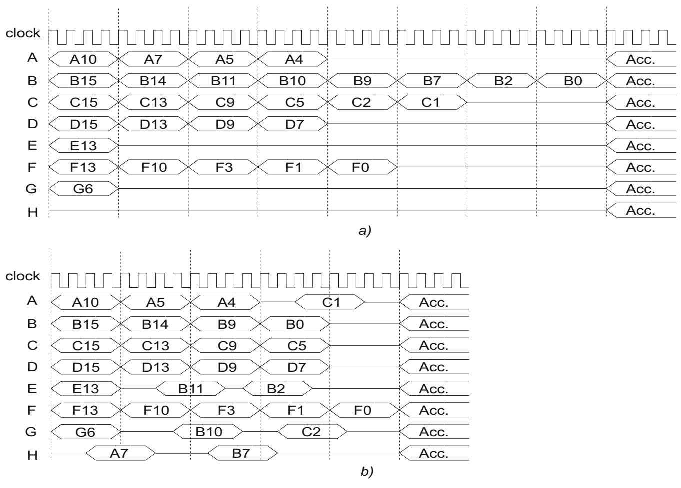


Fig. 1. A, B, ..., H are the agents, horizontal lines indicate no activity, *Acc.* stands for the final summation phase. Alphanumerical labels indicate with the letter the original owner of the triplet-multiplicand pair in process and with the number the triplet number.

evolution of the architecture aimed to introduce cooperation at operation level rather than at sub-operation one.

3.2 Towards an Artificial Swarm System

To overcome the limits shown by the original approach, we rethought the system starting from an abstract representation of it, identifying different layers: environment, communication medium, agents. The *environment* can be conceived as a set of wells that can be filled with data to be processed. In our metaphor, such wells are small local memories with a memory management unit (memory wells). The *communication medium* can be conceived as a set of dedicated pipes locally interconnecting only nearest-neighboring wells. In our metaphor, communication medium is implemented by switches able to perform routing operations without any own initiative. *Hardware agents* form the third layer, where data are processed. At this level cooperative behaviors and computational capabilities are implemented. In our metaphor, a hardware agent is an Arithmetic Logic Unit (ALU) with a smart control implemented as a set of finite state machines (FSMs). A hardware agent stands over a memory well, and consumes only data taken from it. It senses the environment (the workload level of its well and those of the neighboring ones) directing the switches to perform data movements from its well towards adjacent wells with different load. In this way the environment is modified in 2 ways: by means of the direct processing of data by an agent (which decreases the amount of data stored in its well), and by means of explicit data movements. These 2 mechanisms, triggered by the agents, alter the state of the environment in a way that will affect the behavior of other agents, for whom the environment is a stimulus. This is an example of stigmergy, according to the definition given in [20].

This simple approach is based only on local interaction within a regular substrate of simple tiles and apparently is simply a form of workload balance. Instead, it entails several consequences. Indirect interactions do not require neither synchronization on the same communication channel nor handshakes, since environment and agents are different entities. Since agents communication overhead does not increase with the size of the group, stigmergy implicitly allows scalability. The largest number of available tiles can be involved in a task (even in presence of simultaneous multitasking) without any centralized control, thus exploiting at the most the computational platform at any time. This also gives to the architecture less sensitivity to the presence of faulty tiles and faulty agents can be simply excluded by the colony (no reconfiguration is required). Last, software toolchains for compilation/mapping are avoided since data are simply injected into the processing array without any assignment to the hardware resources. Obviously, to achieve these goals, the architecture design must be carefully carried out and several limits arise. We will see in the next Sections how this approach has been converted into an efficient parallel architecture.

4 The Latest Advances in Digital Swarm Architectures

In this Section an architecture implementing the idea and the guidelines presented in Section 3.2 is presented, along with its FPGA implementation [10]. The system is a coprocessing unit for fixed point array operations that can be interfaced with any host processor through an On chip Peripheral Bus (OPB), an embedded system bus developed by IBM. The whole system has been designed and implemented using the Xilinx FPGA EDK 8.1 IDE and simulated by means of ModelSim HDL simulator. The coprocessor model is described in Verilog, at register transfer level (RTL) to ensure the maximum control over the automatic synthesis outcomes. The target hardware device was the XCL4VLX200, belonging to the Virtex-4 family by Xilinx. The Xilinx MicroBlaze, a 32 bit customizable RISC soft core with a MIPS-like instruction set, enhanced with hardware multiplier, divider and barrel shifter, was selected as host processor.

4.1 The Swarm Coprocessor

A block diagram of the swarm coprocessing system presented in this Section is depicted in Fig. 2. The Swarm Coprocessor architecture is sketched in Fig. 3; it is composed of 4 main modules: the Swarm Processing Array (SPA), the Decentralized Column Selector (DCS), the Task Manager and the Coprocessor Data Memory.

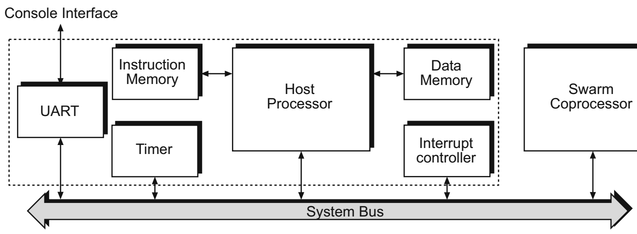


Fig. 2. Block diagram of the Swarm coprocessing system

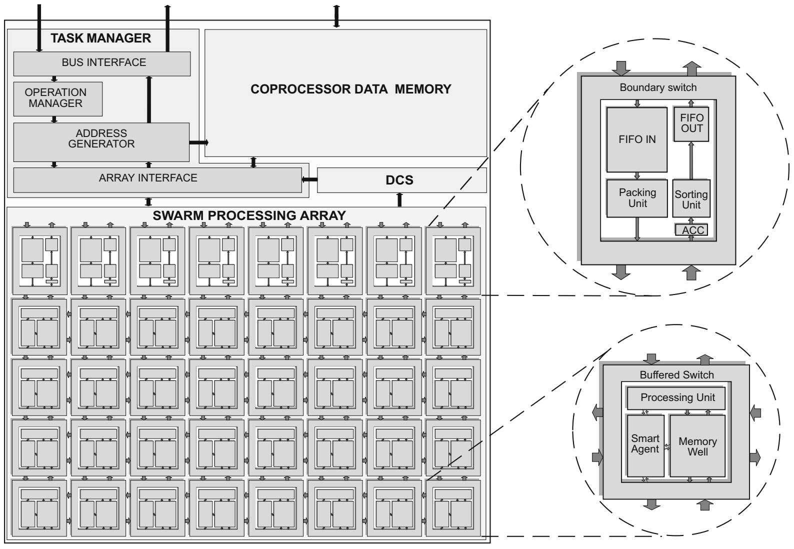


Fig. 3. The Swarm Coprocessor architecture. The computational and border tiles of the Swarm Processing Array have been highlighted.

4.1.1 The Swarm Processing Array

The SPA is the coprocessor core. It is an evolution of the tiled architecture presented in [6], which in turn comes from the original one presented in [27]. It is a 2D grid of locally interconnected undifferentiated tiles (with a row of different tiles hereafter called *border tiles* for data feeding). Even with this regular structure, the adoption of a simple packet switching network able to provide only nearest-neighbors communications allows to reproduce the loose structure of the swarms. The absence of global routing rules, which implies that tiles cannot send packets to other non-neighboring tiles, is a communication limit typical of natural swarm systems, which can be overcome by the adoption of stigmergy. In the implementation presented in this paper, the SPA is composed of 3×8 tiles plus 8 border tiles.

The SPA is conceived for fixed point array processing, so the main goal of every hardware agent is to compute a set of pairwise operations, hereafter called *atomic operations*, between the elements of two arrays. Atomic operations, encapsulated in operation-packets, are the building blocks of a *task* and more than one task can co-exist on the same SPA at the same time. Every tile of the SPA is composed of 4 main blocks: a Local Memory, a Buffered Switch, a Processing Unit and a Smart Agent, as depicted in Figure 3. A hardware agent can be seen as the couple Processing Unit - Smart Agent. The Processing Unit is a sequential ALU characterized by the atomic operation set presented in Table 1. These basic operations are exploited by the Task Manager of the coprocessor to give rise to the final coprocessor operation set.

Table 1. Atomic operations with operands a and b

Operation	Description
MUL	$res = a \times b$
ADD	$res = a + b$
SUB	$res = a - b$
CMP	$res = a > b, a = b, a < b$, and index of occurrence
SHR	$res = b >> a$
SHL	$res = b << a$
MAC	$res_n = res_{n-1} + (a \times b)$
ACC	$res_n = res_{n-1} + (a + b)$

Tasks can return scalars or arrays, respectively composed of one or more result-packets. Tasks composed of one of the first six atomic operations in Table 1 return arrays, the other two return scalars. Both operation-packets and result-packets are 50-bit wide. Operation-packets are represented in Fig. 4, and are composed of the two operands and the operator, the index of the operands (*vector_index*), the displacement of the packet with respect to the original column (*x_disp*) for routing purposes, the operation format (*int_q*), the wrap/saturate option *w/s* and the flag asserting if the packet is a result or not *c/r*. Vector_index field, only useful when the result of the task is a vector, is 8-bit wide thus limiting the size of the vectorial operands involved in this kind of task to 256 elements. Int_q encodes the operands format among integer and fixed point Q15, Q14, Q13. The processing unit in every tile uses this information to apply the proper format performing the required shifts. To preserve the highest precision, tasks involving accumulations are not computed applying truncation and shift at the intermediate results but only after the last accumulation has been performed (in the border tile). In the result-packet (Fig. 5), the value is stored in the place previously taken by the operands, and in the place of the vector_index and opcode a single 11 bits field is present. This field stores the index of the produced result (for tasks returning a vector) or the number of accumulations performed to achieve it (for tasks returning a scalar). This implies that in the latter case the maximum size of the operands is 2048.

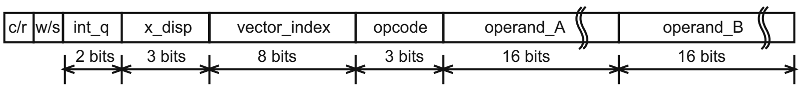


Fig. 4. Format of an operation-packet encapsulating the atomic part of a SPA task

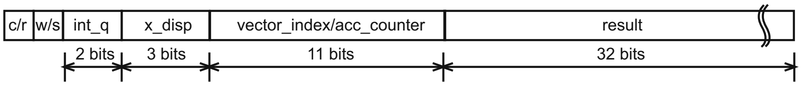


Fig. 5. Format of a result-packet of a SPA task

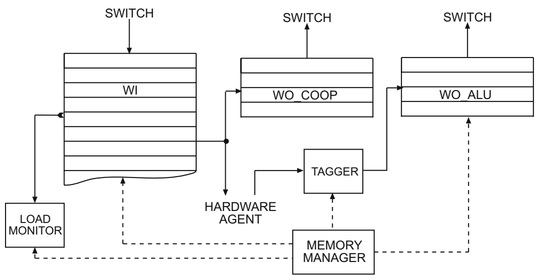


Fig. 6. A simplified representation of the memory well

A detailed description of the switch and the hardware agent is given in [27]. Compared to that implementation, in successive versions we changed the structure of the memory well defining the final structure depicted in Figure 6. Operation-packets to be processed by the hardware agent attached to a memory well are in WI (32 words), whereas result-packets to be returned to the border tiles are placed in the output queue WO_ALU (4 words). The operation-packets to be transferred for load balancing purposes or after fault detection are putted in the WO_COOP queue (4 words). This solution allows a better throughput since it is possible to define priority mechanisms able to avoid the block of the ALU of the hardware agents when the output queue is full and the network is slow due to excessive traffic. A memory manager is responsible for the correct operation of the queues, including tags management and monitor of the actual workload in WI.

Even if the different atomic operations require a different (possibly variable [27]) number of clock cycles to be performed (hereafter named packet weight), it is not correct to consider the workload as the sum of the operation-packet weights in a memory well since the weight of a packet determines its execution speed but should not influence the decision about the number of packets to move for cooperative purposes. This means that the workload must reflect only the number of operation-packets in a memory well and not their weights. Without this assumption, a "bubble effect" could arise, leading heavier tasks to be executed before the light ones, saturating the network [6]. This is an example of an emergent behavior, i.e. an unpredicted behavior of the system caused by the autonomy left to the agents.

As stated in Section 1, there is not enough space to deal with the fault tolerance approach. Shortly, we can say that fault detection is performed by means of both functional tests during inactivity periods and analysis of the operation latency during processing. It is clear that such techniques are not sufficient to provide a good robustness, but they can be enhanced with any of the available Built-in Self Test (BIST) solutions. Once a fault has been detected, the tile is excluded from the array. If the switch is still able to perform data movements, the tile is simply bypassed, otherwise all its ports will be blocked. Simple fault recovery mechanisms have been also implemented to avoid data losses and to ensure the "transparency" of the excluded tile that otherwise would negatively influence the stigmergic interactions [6].

4.1.2 The Decentralized Column Selector

The DCS role is to identify the best column of the SPA for the assignment of a new task ensuring the highest degree of cooperation. The module operates continuously a kind of *tournament selection*, periodically comparing the Available Resources (AR) of the current best column with those of all the other unassigned columns (one at a time). AR takes into account not only the actual load of the columns but also the number of healthy tiles in a column (NHT) [10]. To ensure scalability, the current implementation is distributed so that every agent concurs to the computation of the final value of AR.

Indicating WL_T as the workload of a tile, the available space in the local memory of that tile can be computed as $AR_T = 32 - WL_T$. To take into account the load of the adjacent columns, influencing the cooperation, we can add to AR_T the properly weighted contributions of the horizontally neighboring tiles (AR_{T_L} and AR_{T_R}). This way, for a given column, $AR'_T(n)$ is the quantity propagated by the tile in the n-th row towards the border tile:

$$AR'_T(n) = AR'_T(n-1) + AR_T(n) + 0.25(AR_{T_L}(n) + AR_{T_R}(n)) \tag{1}$$

Since AR is more complex to compute than WL, tiles can equivalently work with WL_T, so that at the border tile of a column arrives:

$$AR'_{col} = 48 \times NHT - \sum_{i=1}^{n_{rows}} WL'_T(i) \tag{2}$$

where $WL'_T(i) = WL_T(i) + 0.25WL_{T_L}(i) + 0.25WL_{T_R}(i)$ with obvious semantics. Central columns can rely on a higher degree of cooperation, being able to exploit up to three columns per side compared to the outermost ones. To take into account this aspect we introduced an offset γ, which takes different values for different columns, as a function of their positions into the SPA, so that $AR_{col} = AR'_{col} + \gamma$.

4.1.3 The Task Manager

The Task Manager module is composed of 4 submodules, as depicted in Fig. 3: the Bus Interface, the Operation Manager, the Address Generator and the Array Interface. Its role is twofold:

- provide an interface between the SPA and the host processor, able to exploit the coprocessor as a peripheral on a standard bus;
- extend the operation set proper of the SPA (Table 1) to the one presented in Table 2, with transparent support for both matrix and vector operations.

Table 2. Swarm Coprocessor operation set. The allowed operands type is specified in the last columns as matrices (M) or vectors (V).

Coprocessor Operation	Function Name	Description *(α is a scalar value)*	Op. type M	 V
Addition	*copr_add*	$RES = A + B$	$\surd$	$\surd$
Scalar addition	*copr_adds*	$RES_i = A_i + \alpha\ \forall i$	$\surd$	$\surd$
Subtraction	*copr_sub*	$RES = A - B$	$\surd$	$\surd$
Scalar subtraction	*copr_subs*	$RES_i = A_i - \alpha\ \forall i$	$\surd$	$\surd$
Compare	*copr_cmp*	$RES_i = 100$ if $A_i > \alpha$ $RES_i = 010$ if $A_i = \alpha\ \forall i$ $RES_i = 001$ if $A_i < \alpha$	$\surd$	$\surd$
Left Shift	*copr_lsh*	$RES_i = A_i << \alpha\ \forall i$	$\surd$	$\surd$
Right Shift	*copr_rsh*	$RES_i = A_i >> \alpha\ \forall i$	$\surd$	$\surd$
Matrix multiplication	*copr_mul*	$RES = A \cdot B$	$\surd$	
Element-wise multiplication	*copr_mulv*	$RES_i = A_i \cdot B_i\ \forall i$		$\surd$
Scalar multiplication	*copr_muls*	$RES = \alpha \cdot A$	$\surd$	$\surd$
Dot product	*copr_mac*	$res = \sum_{i=0}^{N-1}(A_i \cdot B_i)\ \forall i$		$\surd$
Accumulation	*copr_acc*	$res = \sum_{i=0}^{N-1}(A_i + B_i)\ \forall i$		$\surd$
Sample-by-sample FIR filter	*copr_fir*	$res[n] = \sum_{i=0}^{N-1} A[i] \cdot B[n-i]^{\dagger}$		$\surd$

† *with circular buffering.*

In particular, there are 2 interfaces, respectively representing the front end for the host processor and the one for the SPA, namely the *Bus Interface* and the *Array Interface*. The *Operation Manager* is used when a matrix operation is issued to the coprocessor. Since the SPA operation set does not include any matrix operation, they are split by this module into several SPA tasks to be loaded in different columns. For matrix products, an operand is decomposed by rows and the other one by columns, whereas for all the other operations the matrix operands are decomposed by rows. The *Address Generator* is a module dedicated to the generation of the addresses for the coprocessor data memory with different scanning patterns: row/column (used for matrix products), circular (used for FIR filtering), and sequential (used in all the other operations). In Section 4.2 we will see how these modules act in the context of the coprocessor execution flow.

4.1.4 The Coprocessor Data Memory

This RAM memory is internal to the coprocessor module and can be accessed by the host processor via the OPB bus. It is a 3-port memory: one for the host processor and the others to manage the bidirectional data flow to and from the coprocessor. Its size can be changed arbitrarily and in the last implementation presented in literature [10] it is 16K-word, 16-bit plus 2 parity bits per word.

4.2 Swarm Coprocessor Execution Flow

To explain the execution flow onto the Swarm Coprocessor, it is better to divide the software flow by the hardware one.

4.2.1 Software Execution Flow and Programming Model

To perform an operation using the Swarm Coprocessor, the programmer can exploit a library of simple C functions. Two specific data types are defined: *Copr* and *Matrix*. *Copr* is a structure used to identify the Swarm Coprocessor and to provide a control in order to avoid data hazards. It stores the addresses of both Bus Interface and Coprocessor Data Memory, and an array of pointers to the output variables in processing on the coprocessor. From the host processor side, each data in the Coprocessor Data Memory is encapsulated into a *Matrix* structure. It includes the variable (scalar, one-dimensional array, multidimensional array) address in the Coprocessor Data Memory, its number of elements and a flag indicating if this variable is expected to be updated by the coprocessor or not during the current computation.

To perform a sum of 2 matrices with the Swarm Coprocessor, for example, the programmer uses the following library function:

```
int copr_add(Copr * , Matrix * , Matrix * , Matrix * , int );
```

passing in input the pointer to the *Copr* structure and those relative to the operands and the result. The last integer is a control mask used to set the operation parameters needed to fill up the fields of the operation-packets and other information such as the cooperation level. Cooperation level specifies, for the column executing that operation, if the cooperation with other columns is allowed only to be helped (partial stigmergy), helping and being helped (normal stigmergy), or not helping and not being helped (inhibited stigmergy). In this way different priorities are assigned to the operations and it is possible to achieve a deterministic execution time (or at least a near bound) limiting self organization. With the only exception of the first pointer, the other parameters are passed to the Task Manager by means of 4 write operations in 4 memory mapped registers of the Bus Interface.

All the functions work the same way. At first, two controls are performed: the first halts the execution if more operations than the number of SPA columns have been issued and they did not finish yet; the second one checks for data hazards, halting the execution of the function to avoid read after write (RAW) and write after write (WAW) hazards. It should be noted that if the programmer wish to access a variable stored in the Coprocessor Data Memory, the second kind of control should be manually performed in the user code. To this aim, a specific list of data currently in processing is automatically maintained by the software framework in the *Copr* structure. This kind of data hazard problem is due to the fact that the architecture is conceived to allow independent Coprocessor Data Memory accesses by both the processor and the coprocessor, the latter supporting also multitasking, so that more than one task can access the same data area. After such controls, the array of the *Copr* structure is scanned until a free location is available, and the new entry is set, incrementing the number of operations issued.

When an operation is finished, an interrupt service routine (ISR) responds to the interrupt asserted by the coprocessor. It recognizes which operation finished reading the address of the result from the Bus Interface and then updating the list of the data currently in processing. This software infrastructure, which is also needed for the analysis of dependencies to avoid data hazards, rests on a set of drivers providing basic OPB functions. They are required since, for instance, to access a variable stored in the Coprocessor Data Memory from the host processor no direct access through pointers or addresses is allowed.

4.2.2 Hardware Execution Flow Inside the Coprocessor

At a hardware level, the operation issued by the processor is received in the Bus Interface in the aforementioned 4 registers, and then transferred to a FIFO accessible by the Operation Manager, which identifies the opcode and defines which task (or set of tasks) must be generated. Once generated the tasks, the Address Generator provides the scanning patterns to access the variables in the Coprocessor Data Memory, and the Array Interface transfers the task to the column identified by the DCS. The Array Interface fills the FIFO_IN memory in the boundary tile of the selected column (2×1024 16-bit locations). In turn, the boundary tile can start creating the operation-packets with its Packing Unit (Figure 3), sending them to the first tile below in the SPA, as soon as the first data are available.

Tasks are loaded by columns: the memory well of the northernmost tile is filled up; when it is full the second one is filled up and so on, moving towards the southern border of the SPA (Figure 3). Stigmergy is inhibited in the column in charge until either the task is completely loaded or the column is full; then the cooperation level value determines the stigmergic behavior for that column. During load, tiles start transforming their operation-packets into result-packets, which are sent towards the border tile. Stigmergy works spreading the workload and thus involving in the task not only the tiles of that column but also those of the neighboring ones, within a range of 3 columns per side. result-packets received by the boundary tile can be either sorted in the FIFO_OUT memory (if the task produced an array) or partially accumulated (if the task must return a scalar). In the last case the result is stored in the FIFO_OUT memory (1×256 16-bit locations) only when the task is finished. It should be noted that the accumulations performed by the boundary tiles for cumulative tasks are less than the required number, since computational tiles perform the allowed accumulations during processing, sending the result-packet to the border tile only when their memories are empty or when an operation-packet from another task must be processed.

When the task is finished, the boundary tile informs the Array Interface which in turn is responsible for the selection of the column enabled to transfer its results from the FIFO_OUT to the Coprocessor Data Memory, using a priority criteria that is based on the specific operations in processing. When the results transfer ends, an interrupt is asserted to the processor and the address of the result is placed on a FIFO provided by the Bus Interface, ready to be read by the host processor.

5 Architecture Evaluation

To evaluate a digital architecture, area and frequency parameters are usually taken into account. We must say that these 2 parameters do not fit well with the FPGA implementation presented in this Chapter, since the system was rather conceived for standard cells synthesis. This means that some choices, as the one to implement all the memory wells as register banks rather than RAMs, are not suited for a FPGA realization, where they lead to waste hardware resources. A synthesis of the SPA on standard cell CMOS $0.13\mu m$ technology resulted in an operating frequency of 900KHz for the global system, with the Processing Units of the tiles running at 450KHz. The architecture is truly scalable, i.e. the area grows linearly with the number of tiles without side effects on the operating frequency. Every tile requires about 44K equivalent gates, with the 78% of this area occupied by the memories.

To evaluate the performance of the system, we had recourse to HDL simulations. To perform such simulations, UART and timer modules were needed (Figure 2). The UART was used as standard output in order to print the results onto a console, using a *named pipe*-based interface between the UART simulation model and the console [2]. The timer was used to perform a cycle accurate profiling, counting the clock cycles needed to complete a specific operation or an entire application.

5.1 *Cooperative Processing at a Glance*

We said above that the most evident effect of the proposed cooperative approach is workload balance. We can visually analyze the effect of the cooperation on the execution of a single task (a dot product between two 2048-element integer vectors) into the SPA looking at Figure 7. The workload spreading over time inside the array, by columns, is represented in false colors with or without stigmergy.

To produce this graphical representation, during a simulation we monitored the workload of every tile at every clock tick. Then we obtained a single value per clock cycle for every column by summing the contribute of each tile. Performance improvements achievable by means of stigmergy and self organization into the SPA of the Swarm Coprocessor are clearly visible. It is worth to note that the task

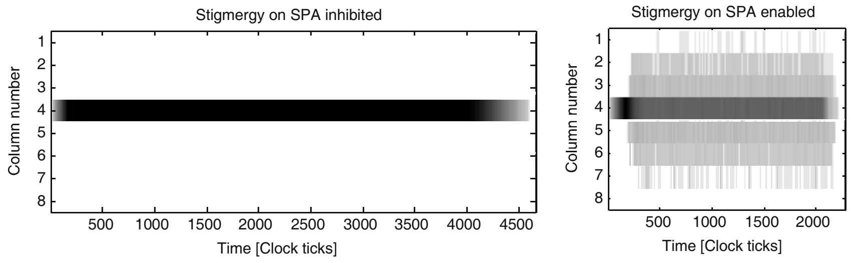


Fig. 7. Non-cooperative and cooperative execution of a *copr_mac* into the Swarm Coprocessor

management is completely decentralized and does not require any configuration, i.e. the system resorts to self organization to provide a consistent speed-up.

This behavior is adaptive since, if multiple tasks are in execution on the coprocessor at the same time, cooperative behaviors will progressively reduce their effect on the overall processing time. The case with two tasks (*copr_mac* between two 130-element arrays) is depicted in Figure 8, where different snapshots have been sampled at the same instants of time for the execution enabling or not stigmergy. The height of the bars represents the workload of the various tiles. Looking at the time axis it is possible to note that cooperation remarkably reduces the computational latency to 47% of the case without cooperation [6].

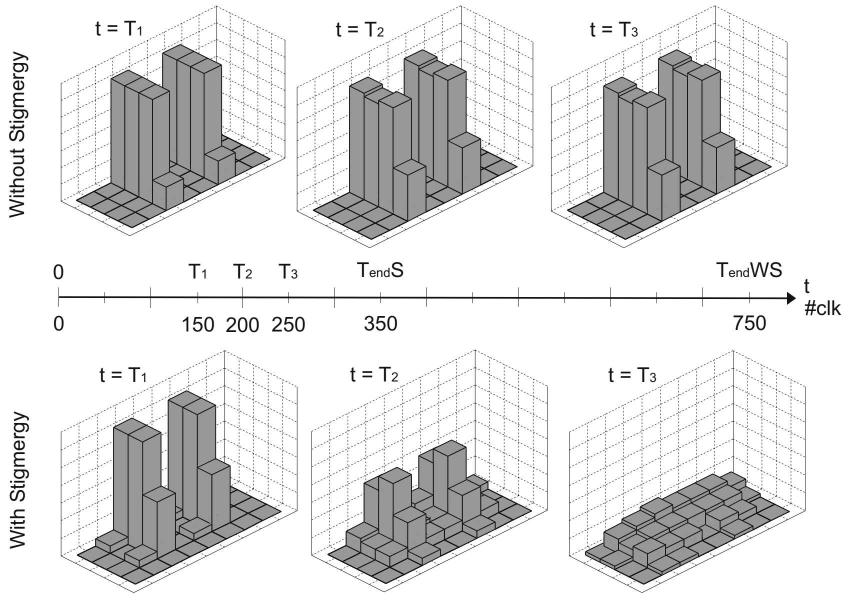


Fig. 8. Workload distribution evolution in different instants of time within the SPA when 2 *copr_mac* tasks are in execution, enabling or not stigmergy

5.2 Single-Operation Profiling

One way to evaluate the performance of the Swarm Coprocessor is to load it with a single task, varying the size of the arrays. For each configuration, the execution on the coprocessor was profiled enabling or disabling stigmergy on the SPA, and then running the task on the host processor for a performance comparison. In the last case operands and results have been stored in the host processor data memory rather than on the Coprocessor Data Memory to improve the processor performances reducing the access latency. Even with these attentions, the comparison is not fair because the coprocessor exploits a larger number of computational units to perform the same task, even if not explicitly preprogrammed by the user. Alternatively enabling and disabling stigmergy it is possible to achieve approximately two bounds for the performance on the coprocessor. In this way, the performance in any load condition will fall within the region limited by such curves in the latency vs. task size plane.

Figure 9 presents the results of a small set of tasks, due to the limited space: 2 of them return an array (*copr_add* and *copr_mulv*), the other 2 a scalar (*copr_acc* and *copr_mac*). The Swarm Coprocessor achieves a speed-up of 11 for the tasks with vectorial result and of 30 on those with scalar result. This difference is motivated in the light of considerations about the size of the tasks and the required bandwidth. As a matter of fact, cumulative tasks can use larger operands than vectorial ones (2048 rather than 256) for the reasons presented in Section 4.1.1. In this way they can exploit at the most the cooperative approach used by the Coprocessor to carry

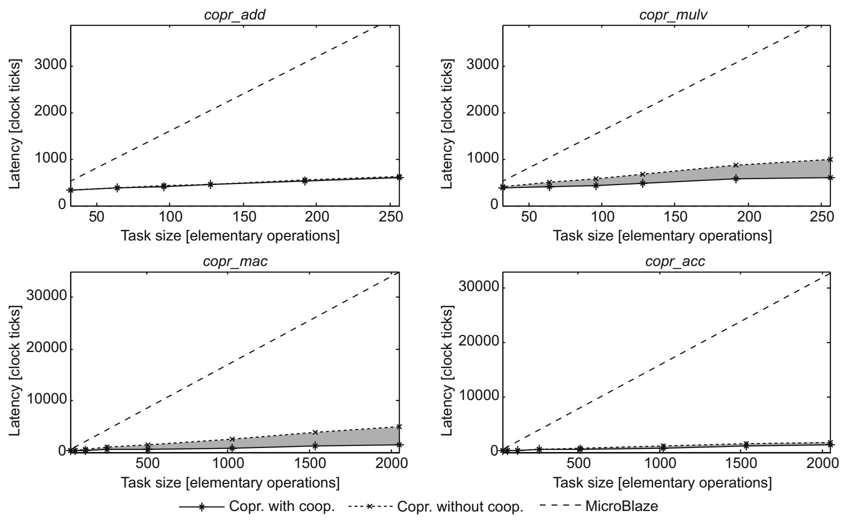


Fig. 9. Comparison of the latencies to perform some tasks on vectors of integers on the MicroBlaze processor (best configuration) and on the processor-coprocessor system, enabling or not the stigmergy (and then cooperation) in the SPA

out the required processing. Furthermore cumulative operations generate less result-packets compared to the vectorial ones, and this reduces bandwidth requirements avoiding the saturation of the network.

Figure 9 also shows that latency grows linearly with the task size for both the coprocessor and the processor. OPB communication and interrupt service overheads are quantifiable in $240-400$ clock cycles depending on the number of tasks in execution on the coprocessor. For this reason the processor performs better than the Swarm Coprocessor for tasks below the 20 atomic operations.

5.3 Simple Applications Profiling

The Swarm Coprocessor was also tested on simple real-world applications (matrix multiplications, image threshold, weighted sum of vectors, matrix square norm), comparing its performance to that of the host processor, considering their best operating conditions. The results are presented in Table 3. Beyond matrix multiplication, which is automatically split into $N \times M$ tasks by the Task Manager, the other applications require a manual decomposition in tasks by the programmer. This is common to several DSP optimization procedures.

Table 3. Latency comparison using or not the Swarm Coprocessor on simple applications

	Matrix Multiplication			**Simple applications**		
	$A_{n\times192}\cdot B_{192\times n}$			W. sum	Image th.	Matrix $\|\|\cdot\|\|^2$
	$n=8$	$n=16$	$n=32$	(1×256)	(16×100)	(8×1024)
Host + Copr.	769	3331	20868	1564	6046	5030
Host	11717	87813	678917	4630	24612	57400

Image threshold can be decomposed into 3 tasks: a comparison with the threshold (*copr_cmp*), a shift by 2 of the previously achieved result (*copr_rsh*, see Table 2), and the multiplication of this result for the scalar standing for the maximum in the grey scale (*copr_muls*). Since every task is executed on the outcome of the previous one, after every assignment the execution on the processor is frozen, hence leading to an inefficient scheduling.

The weighted sum of vectors can be better parallelized since the application of the two coefficients of the linear combination to the vectorial operands (*copr_muls*) can be accomplished in parallel, whereas the final summation cannot (*copr_add*).

The squared norm of a matrix is decomposed into N *copr_mac* tasks, one for each row, and then the final accumulation (*copr_acc*) is performed. This benchmark is more parallelizable than the others if $N > 2$.

It should be noted how the intrinsic parallelism of the application greatly influences the performance. As a matter of fact, a greater performance improvement can be achieved on matrix operations, where the overhead for the assignment of the task is scarce with respect to the amount of tasks autonomously created in the Swarm

Coprocessor by the Operation Manager. This leads the matrix product, in spite of the single operation assigned, to produce a speed up of 36. Beyond matrix multiplications, matrices element-wise operations are always split in several (*N*, being *N* the number of rows of the two matrices) tasks, so that even in this case the overhead for operation assignment is compensated by the number of tasks automatically generated by the Swarm Coprocessor front-end.

The coprocessor performance is sufficient to execute also real-time operations. In these cases, stigmergy should be inhibited to avoid the non-determinism in the processing time due to the cooperation between columns. However, cooperation is still exploited within the same column. Performance was analyzed on FIR filters with 512 and 1024 taps and a variable number of channels (1, 4, 8). These tasks are supported by a circular addressing in the Task Manager Address Generator. The maximum sampling rates achievable with Swarm Coprocessor range from 71KHz (for 8 channels, 1024 taps) to 368KHz (for 1 channel, 512 taps), respectively 9.2 and 20.3 times the sampling rate achievable with the host processor alone. For such estimates we are referring to the operating frequencies of the standard cell implementation.

5.4 Multitasking Behavior

We stated above that the performance of the Swarm Coprocessor grow with the parallelism of the application. This is true for the simple applications presented in Section 5.3, but mostly for multitasking applications, where the different computations can be assigned at the same time to the coprocessor. This is clearly visible in Figure 10, where the length of the bars is a measure of the latency of a task: black bars are referred to the case without any cooperation, grey ones to the case with cooperation enabled. When the tasks concentration is denser, performance improvements caused by cooperation are less appreciable. The same holds for the very small tasks too. It can be noted that the task assignment to the different columns

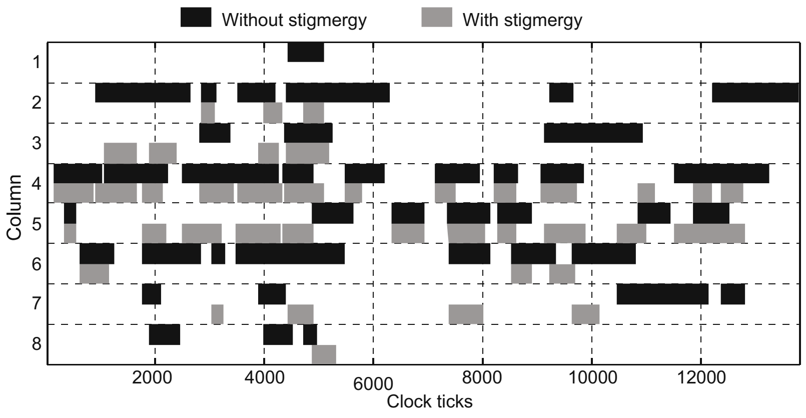


Fig. 10. Multitasking within the SPA enabling (in grey) or disabling (in black) stigmergy

changes when stigmergy is enabled since the DCM chooses the best column at run time, and the central columns are preferred to the outermost (if available) due to the parameter γ that takes into account the potential cooperation level associated to the position of the column (Section 4.1.2). It is worth to note that, in large architectures, it would be impossible to efficiently perform similar run-time analyses taking the relative decisions in a centralized way.

6 Conclusions

In this Chapter we presented and analyzed a new approach to the design of digital architectures for parallel distributed on-chip processing. It is based on the exploitation of Swarm Intelligence, a bio-inspired multi-agent paradigm that, compared to other agent-based approaches, focuses on the simplicity of the agents and their communication strategies. Our interpretation of this paradigm takes advantage of these aspects to be successful in the realization of a multi-agent model based on “hardware agents” with very low complexity compared to microprocessors executing “agent codes”. Agents are able to achieve a global coordination in the execution of multiple tasks in parallel exploiting cooperative processing without any centralized control. This fully-decentralized control approach, which does not require any configuration or pre-programming, allows to avoid the development of complex compilation/mapping tools. All the decisions are taken by the agents at run time, and the programmer’s job is limited to the exploitation of a set of simple C libraries, as common to many other code-optimization problems.

The usage of indirect interactions (stigmergy) for communications within the architecture it is possible to achieve good performance enhancing concurrency (and then parallelism), locality and intrinsically supporting fault tolerance. Even if this chapter does not deal with fault tolerance issues, we demonstrated the potentiality of the proposed solution with respect to this topic [6]. As in natural swarms, the effect of faults in some tiles is a slight reduction of the system performance, sometimes too small to be appreciable. It should be noted that fault detection and recovery (both at agent and colony level) is supported without any reconfiguration or exploitation of spare resources.

The main limits of the current architecture are the reduced set of supported operations and the impossibility to deal with some complex tasks exposing data dependencies. As a matter of fact, some algorithms require the definition of an execution flow graph that could not be mapped onto the proposed architecture. The current research work aims to extend the Swarm model to address this issue, allowing the identification of the agents without direct addressing and extending the operation set. Furthermore, a floating point version of the system is currently under testing.

Even after 5 years from the first exploration in this field, only a few steps have been taken towards the systematization of this new approach, and the contributes of the scientific community are needed to deeply investigate it. When emerging technologies will require novel design strategies, the Swarm Intelligence model could be an alternative solution, in the era of decentralization, for parallel integrated computing.

References

1. Ambric Inc., http://www.ambric.com
2. MSDN Library: Named Pipes, http://msdn2.microsoft.com/en-us/library/aa365590.aspx
3. PicoChip Designs Ltd., http://www.picochip.com
4. Tilera corporation, http://www.tilera.com
5. Agarwal, V., Hrishikesh, M., Keckler, S.W., Burger, D.: Clock rate versus IPC: The end of the road for conventional microarchitectures. In: Proc. of the 27th Annual International Symposium on Computer Architecture (2000)
6. Angius, G., Manca, C., Pani, D., Raffo, L.: Cooperative VLSI tiled architectures: Stigmergy in a swarm coprocessor. In: Dorigo, M., Gambardella, L.M., Birattari, M., Martinoli, A., Poli, R., Stützle, T. (eds.) ANTS 2006. LNCS, vol. 4150, pp. 396–403. Springer, Heidelberg (2006)
7. Bartolini, S., Giorgi, R., Martinelli, E., Popovic, Z.: Recent proposals for tiled architectures. In: Poster Abstract of the HiPEAC ACACES-2005 Summer School, pp. 47–50. Academia Press (2005)
8. Beni, G.: The concept of cellular robotic system. In: Proc. 1988 IEEE Int. Symp. on Intelligent Control, Los Alamitos, CA (1988)
9. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence, From Natural To Artificial Systems. Oxford University Press, Oxford (1999)
10. Busonera, G., Carucci, S., Pani, D., Raffo, L.: Self-organization on silicon: system integration of a fixed-point swarm coprocessor. In: Proc. Nature Inspired Cooperative Strategies for Optimization (NICSO 2007), pp. 149–158 (2007)
11. Caro, G.D., Ducatelle, F., Gambardella, L.M.: Anthocnet: an ant-based hybrid routing algorithm for mobile ad hoc networks. In: Yao, X., Burke, E.K., Lozano, J.A., Smith, J., Merelo-Guervós, J.J., Bullinaria, J.A., Rowe, J.E., Tiňo, P., Kabán, A., Schwefel, H.-P. (eds.) PPSN 2004. LNCS, vol. 3242, pp. 461–470. Springer, Heidelberg (2004)
12. Chen, H., Qian, D., Wu, W., Cheng, L.: Swarm intelligence based energy balance routing for wireless sensor networks. In: Second International Symposium on Intelligent Information Technology Application, IITA 2008, vol. 2, pp. 811–815 (2008)
13. Colorni, A., Dorigo, M., Maniezzo, V.: Distributed optimization by ant colonies. In: Proc. of the First European Conference on Artificial Life, Paris, France, pp. 134–142 (1992)
14. Dally, W., Lacy, S.: VLSI architecture: past, present, and future. In: Proc. 20th Anniversary Conference on Advanced Research in VLSI, Las Vegas, USA, pp. 232–241 (1999)
15. Dixit, S., Sarma, A.: Advances in self-organizing networks. IEEE Communications Magazine 43(7), 76–77 (2005)
16. Dorigo, M., Maniezzo, V., Colorni, A.: The Ant System: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics 26(1), 29–41 (1996)
17. Grassé, P.: La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. la theorie de la stigmergie: Essai d'interpretation des termites constructeurs. Ins. Soc. 6, 41–83 (1959)
18. Gruau, F., Lhuillier, Y., Reitz, P., Temam, O.: Blob computing. In: Computing Frontiers 2004 ACM SIGMicro (2004)
19. Kassabalidis, I., El-Sharkawi, M.A., Marks II, R.J., Arabshahi, P., Gray, A.: Swarm intelligence for routing in communication networks. IEEE GlobeComm (2001)
20. Kennedy, J., Eberhart, R., Shi, Y.: Swarm Intelligence. Morgan Kaufmann Academic Press, San Francisco (2001)

21. Kube, C.R.: Collective robotics: From local perception to global action. Ph.D. thesis, Dept. of Computer Science, Univ. of Alberta, Edmonton (1997)
22. Kuntz, P., Layzell, P.: An ant clustering algorithm applied to partitioning in VLSI technology. In: Proc. of the 4th European Conference on Artificial Life, pp. 417–424 (1997)
23. Liu, Y., Passino, K.: Swarm intelligence: Literature overview. Tech. rep., The Ohio State University, Columbus, OH, Internal Report (2000)
24. Martinoli, A., Easton, K.: Modeling swarm robotic systems. In: Proc. of the Eight Int. Symp. on Experimental Robotics ISER 2002, Sant'Angelo d'Ischia, Italy (2002)
25. Pani, D., Raffo, L.: A swarm intelligence based VLSI multiplication-and-add scheme. In: Yao, X., Burke, E.K., Lozano, J.A., Smith, J., Merelo-Guervós, J.J., Bullinaria, J.A., Rowe, J.E., Tiňo, P., Kabán, A., Schwefel, H.-P. (eds.) PPSN 2004. LNCS, vol. 3242, pp. 362–371. Springer, Heidelberg (2004)
26. Pani, D., Raffo, L.: A VLSI multiplication-and-add scheme based on swarm intelligence approaches. In: Dorigo, M., Birattari, M., Blum, C., Gambardella, L.M., Mondada, F., Stützle, T. (eds.) ANTS 2004. LNCS, vol. 3172, pp. 13–24. Springer, Heidelberg (2004)
27. Pani, D., Raffo, L.: Stigmergic approaches applied to flexible fault-tolerant digital VLSI architectures. Journal of Parallel Distributed Computing, Special Issue on Parallel Bioinspired Algorithms 66(8), 1014–1024 (2006)
28. Parhi, K.: VLSI Digital Signal Processing Systems - Design and Implementation. Wiley-Interscience, Hoboken (1999)
29. Rabbah, R.M., Bratt, I., Asanovic, K., Agarwal, A.: Versatile tiled-processor architectures: the Raw approach. In: Proc. of the Eighth Annual High Performance Embedded Computing Workshop (HPEC) (2004)
30. Resnick, M.: Turtles, Termites, and Traffic Jams - Explorations in Massively Parallel Microworlds. MIT Press, Cambridge (1997)
31. Rotenberg, E.: AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In: Proc. of the 29th IEEE International Symposium on Fault-Tolerant Computing (FTCS-29), pp. 84–91 (1999)
32. Sahin, E., Labella, T., Trianni, V., Deneubourg, J.L., Rasse, P., Floreano, D., Gambardella, L., Mondada, F., Nolfi, S., Dorigo, M.: SWARM-BOT pattern formation in a swarm of self-assembling mobile robots. In: Proc. of the IEEE International Conference on Systems, Man and Cybernetics, Hammamet, Tunisia (2002)
33. Schoonderwoerd, R., Holland, O.E., Bruten, J.L., Rothkrantz, L.J.M.: Ant-based load balancing in telecommunications networks. Adaptive Behavior (2), 169–207 (1996)
34. Socha, K., Dorigo, M.: Ant colony optimization for continuous domains. European Journal of Operational Research 185(3), 1155–1173 (2008)
35. Taillard, E.: Ant systems. Tech. Rep. IDSIA-05-99, IDSIA, Lugano (1999)
36. Trianni, V., Nolfi, S., Dorigo, M.: Evolution, self-organization and swarm robotics. In: Blum, C., Merkle, D. (eds.) Swarm Intelligence. Natural Computing Series, ch. 5, pp. 43–85. Springer, Heidelberg (2008)

Large Scale Bioinformatics Data Mining with Parallel Genetic Programming on Graphics Processing Units

William B. Langdon

Abstract. A suitable single instruction multiple data GP interpreter can achieve high (Giga GPop/second) performance on a SIMD GPU graphics card by simultaneously running multiple diverse members of the genetic programming population. SPMD dataflow parallelisation is achieved because the single interpreter treats the different GP programs as data. On a single 128 node parallel nVidia GeForce 8800 GTX GPU, the interpreter can out run a compiled approach, where data parallelisation comes only by running a single program at a time across multiple inputs.

The RapidMind GPGPU Linux C++ system has been demonstrated by predicting ten year+ outcome of breast cancer from a dataset containing a million inputs. NCBI GEO GSE3494 contains hundreds of Affymetrix HG-U133A and HG-U133B GeneChip biopsies. Multiple GP runs each with a population of five million programs winnow useful variables from the chaff at more than 500 million GPops per second. Sources available via FTP.

1 Introduction

Due to their speed, price and availability, there is increasing interest in using mass market graphics hardware (GPUs) for scientific applications. Since our initial experiments GPU development has continued apace. For example, AMD has launched its 800 $\times$ 750MHz processor ATI Radeon HD 4870. Whilst almost simultaneously nVidia launched its 240 $\times$ 1296MHz GTX 280 GPU. Both claim to deliver about one Tetraflop at a cost of a few hundred dollars.

The next section will describe scientific and engineering computing on GPUs. (Known as GPGPU). So far there are a few reported successful applications of GPUs to Bioinformatics. These will be described in Section 3. In Section 4 we will

William B. Langdon
King's College, London

describe one where genetic programming [24] is used to datamine a small number of indicative mRNA gene transcript signals from breast cancer tissue samples taken during surgery. Section 5 describes how we run GP [18, 2, 27, 45] in parallel on a GPU. Whilst the rest of Section 5 (i.e. 5.1 and 5.2) and Section 6 describe the medical problem and the way a powerful GPU [29, 26] simultaneously picks three of the million mRNA measurements available and finds a simple non-linear combination of them which predicts long term outcomes at least as well as DLDA, SVM and KNN using seven hundred measurements [37].

2 Using Games Hardware GPUs for Science

Owens *et al.* have recently surveyed scientific and engineering applications running on mass market graphics cards (known as general purpose computing on GPUs, i.e. GPGPU) [42, 43]. Whilst there is increasing interest, so far both Bioinformatics and computational intelligence are under represented. As with other GPGPU applications, the drivers are: locality, convenience, cost and concentration of computer power. Indeed the principle manufactures (nVidia and ATI) claim faster than Moore's Law increase in performance (e.g. [11, p4]). They suggest that GPU floating point performance will continue to double every twelve months, rather than the 18-24 months observed for electronic circuits in general [38] and personal computer CPUs in particular. Indeed the apparent failure of PC CPUs to keep up with Moore's law in the last few years [42, p890]. makes GPU computing even more attractive. Even today's top of the range GPU greatly exceed the floating point performance of their host CPU. This speed comes at a price.

GPUs provide a restricted type of parallel processing, often referred to a single instruction multiple data (SIMD) or more precisely single program multiple data (SPMD). Each of the many processors simultaneously runs the same program on different data items. See Figure 1. Being tailored for fast real time production of interactive graphics, principally for the computer gaming market, GPUs are tailored to deal with rendering of pixels and processing of fragments of three dimensional scenes very quickly. Each is allocated a processor and the GPU program is expected to transform it into another data item. The data items need not be of the same type. For example the input might be a triangle in three dimensions, including its orientation, and the output could be a colour expressed as four floating point numbers (RGB and alpha). Indeed vectors of four floats can be thought of as the native data type of current GPUs. RapidMind's software translates other data types to floats when it transfers it from the CPU's memory to the GPU and back again when results are read back. Note integer precision may only be 24 bits, however GPUs will soon support 64 bits.

Typical GPUs are optimised so that programs can read data from multiple data sources (e.g. background scenes, placement of lights, reflectivity of surfaces) but generate exactly one output. This parallel writing of data greatly simplifies and

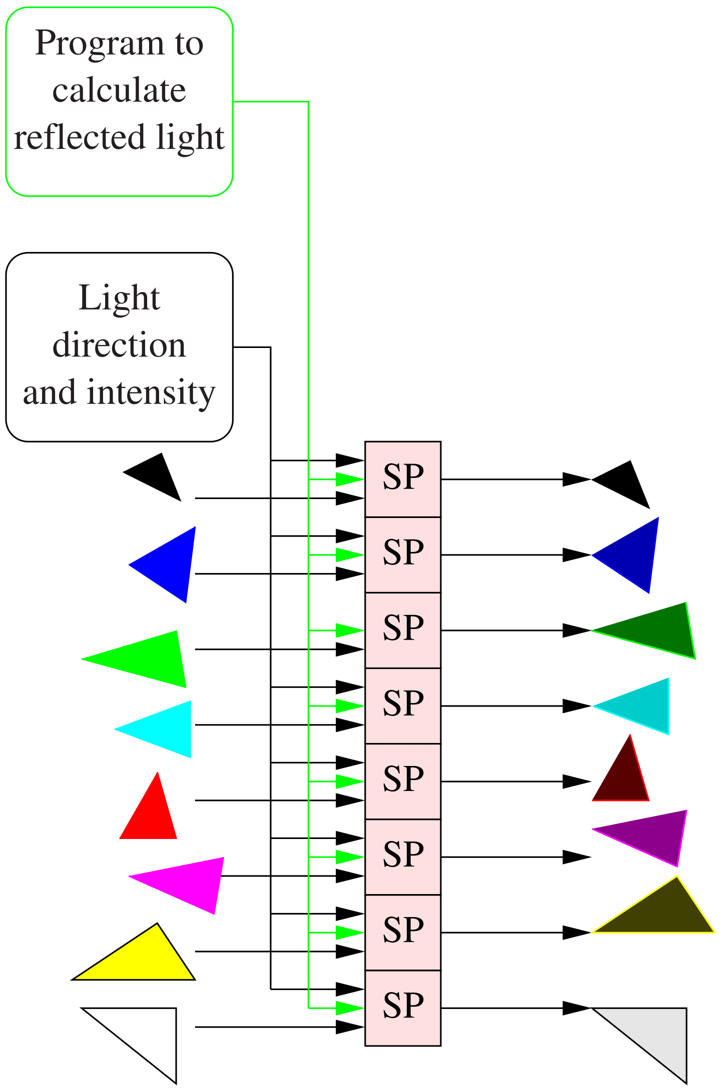

Program to calculate reflected light
Light direction and intensity
SP
SP
SP
SP
SP
SP
SP
SP

Fig. 1. An example of SIMD parallel processing. The stream processors (SP) simultaneously run the same program on different data and produce different answers. In this example each programs has two inputs. One describes a triangle (position, colour, nature of its surface: matt, how shiny). The second input refers to a common light source and so all SP use the same value. Each SP calculates the apparent colour of its triangle. Each calculation is complex. The stream processors use the colour of the light, angles between the light and its triangle, direction of its triangle, colour of its triangle, etc.

speeds the operation of the GPU. Even so both reading and writing from memory are still bottlenecks. This is true even though GPUs usually come with their own memory and memory caches. (The nVidia 8800 comes with 768Mbytes). Additionally data must be transfered to and from the GPU. Even when connected to the CPU's RAM via PCI, this represents an even narrower bottle neck. Faster hardware (e.g. PCI Express x16) is available for some PC/GPU combinations. However this does not remove the bottle neck. CPU–GPU communication can also be delayed by the operating system check pointing and rescheduling the task.

The manufactures' publish figures claiming enormous peak floating point performance. In practise such figures are not obtainable. A more useful statistic is often how much faster an application runs after it has been converted to run on a GPU. However the number of GP operations per second (GPops) should allow easier comparison of different GP implementations.

Many scientific applications and in particular Bioinformatics applications are inherently suitable for parallel computing. In many cases data can be divided into almost independent chunks which can be acted upon almost independently. There are many different types of parallel computation which might be suitable for Bioinformatics. Applications where a GPU might be suitable are characterised by:

- Maximum dataset size $\approx 10^8$
- Maximum dataset data rate $\approx 10^8$ bytes/second
- Up to 10^{11} floating point operations per second (FLOPs)
- Applications which are dominated by small computationally heavy cores. I.e. a large number of computations per data item.
- Core has simple data flow. Large fan-in (but less than sixteen) and simple data stream output (no fan-out).

Naturally as GPUs become more powerful these figures will change.

In some cases, it might be possible to successfully apply GPUs to bigger problems. For example, a large dataset might be broken into smaller chunks, and then each chunk is loaded one at a time onto the GPU. When the GPU has processed it, the next chunk is loaded and so on, until the whole dataset has been processed. The time spent loading data into (and results out of) each GPU may be important. If the application needs a data rate of 100Mbyte/second we must consider how the data is to be loaded into a personal computer at this rate in the first place. Alternatively it may be possible to load data from a scientific instrument directly connected to the GPU.

nVidia say their GeForce 8800 (Fig. 2) has a theoretical upper limit of 520 GFLOPS [39, p36], however we obtained about 30 GFLOPS in practice. Depending on data usage (cf. Section 7), it appears that 100 GFLOPS might be reached in practise. While tools to support general purpose computation on GPUs have been greatly improved, getting the best from a GPU is still an art. Indeed some publications claim a speed up of only 20% (or even less than one) rather than 7+, which we report.

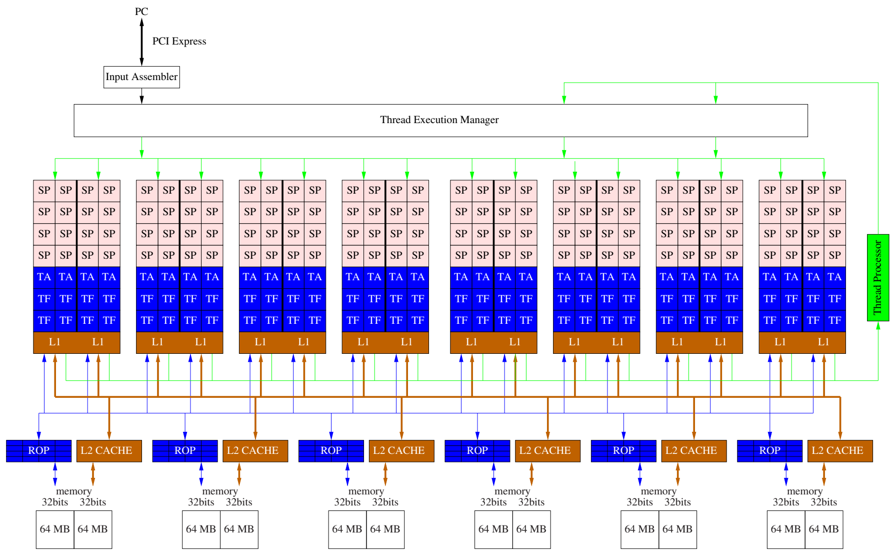


Fig. 2. nVidia 8800 Block diagram. The 128 1360 MHz Stream Processors are arranged in sixteen blocks of eight. Blocks share 16 KB memory (not shown), an 8/1 KB L1 cache, four Texture Address units and eight Texture Filters. The 6×64 bit bus (brown) links off chip RAM at 900 (1800) MHz [39, 40]. There are 6 Raster Operation Partitions (ROP).

3 GPUs in Bioinformatics and Computational Intelligence

As might be expected GPUs have been suggested for medical image processing applications for a few years now. However we concentrate here on molecular bioinformatics. We anticipate that after a few key algorithms are successfully ported to GPUs, within a few years Bioinformatics will adopt GPUs for many of its routine applications. As might be expected, early results were mixed.

Charalambous *et al.* successfully used a relatively low powered GPU to demonstrate inference of evolutionary inheritance trees (by porting RAxML onto an nVidia) [4]. However a more conventional MPI cluster was subsequently used [50]. Sequence comparison is the life blood of Bioinformatics. Liu *et al.* ran the key Smith-Waterman algorithm on a high end GPU [31]. They demonstrated a reduction by a factor of up to sixteen in the look up times for most proteins. Smith-Waterman has also been ported to the Sony PlayStation 3 [54] and the GeForce 8800 (CUDA) [35]. Schatz *et al.* also used CUDA to port another sequence searching tool (MUMmer) to another G80 GPU and obtained speed ups of 3–10 when matching short DNA strands against much longer sequences [49]. By breaking queries into GPU sized fragments, they were able to run short sequences (e.g. 50 bases) against a complete human chromosome. Gobron *et al.* used OpenGL on a high end GPU to drive a cellular automata simulation of the human eye and achieved real-time processing of webcam input [13]. GPUs have also been used in medical engineering.

E.g. a GeForce 8800 provided a 15-20 fold speedup, improving the haptic response of a real time interactive surgery simulation tool [32]. Dowsey *et al.* wrote 2D gel electrophoresis image registration code in Cg ("C for graphics") so that it could be off loaded onto an nVidia GPU [7].

The better GPU applications may claim speed ups of a factor of ten or so, however the distributed protein folding system folding@home obtains sixty times as much free computation per donated GPU as it does per donated CPU [42, p983]. The same authors also claim almost a 3600 fold speed up on a biomolecule dynamics simulation, albeit at the cost of using four FX 5600 GPUs [42, p995].

Computational intelligence applications of GPUs have included artificial neural networks (e.g. multi layer perceptrons and self organising networks [34]), genetic algorithms [12] and a few genetic programming experiments [30, 33, 36, 8, 47, 16, 14, 17, 15, 5, 21, 26, 48, 52, 1].

Most GPGPU applications have only required a single graphics card, however Fan *et al.* have shown large GPU clusters are also feasible [9]. In 2008 the first computational intelligence on GPU special session (CIGPU-2008) was held in Hong Kong [53]. It is anticipated that this will become an annual event. As Owens [43] makes clear games hardware is now breaking out of the bedroom into scientific and engineering computing.

4 Gene Expression in Breast Cancer

Miller *et al.* describes the collection and analysis of cancerous tissue from most of the women with breast tumours from whom samples were taken in the three years 1987–1989 in Uppsala in Sweden [37]. Miller's primary goal was to investigate p53, a gene known to be involved in the regulation of other genes and implicated in cancers. In particular they studied the implications of mutations of p53 in breast cancer. The p53 genes of 251 women were sequenced so that it was known if they were mutant or not. Affymetrix GeneChips (HG-U133A and HG-U133B) were used to measure mRNA concentrations in each biopsy. Various other data were recorded, in particular if the cancer was fatal or not.

Affymetrix GeneChips estimate the concentration of strands of messenger RNA by binding them to complementary DNA itself tied to specially treated glass slides. GeneChips are truly amazing. When working well they can measure the activity, in terms of mRNA concentration, of almost all known human genes in one operation. Each of the two types of GeneChips used contained more than half a million DNA probes arranged in a 712×712 square $(12.8\text{mm})^2$ array. (Current designs now exceed five million DNA probes on the same half inch square array.) Obviously such tiny measuring devices are very subject to noise and so between eleven and twenty readings are taken per gene. In fact each reading is duplicated with a control which differs only by its central DNA base. These controls are known as mismatch MM probes.

There has been considerable debate about the best way of converting each of the eleven or more pairs of readings into a single value to represent the activity of a gene.

Also in more recent designs (e.g. exon arrays), Affymetrix have replaced the MM probes of each pair with general area control probes. Miller *et al.* used Affymetrix' MAS5 program. MAS5 uses outlier detection etc. to take a robust average of the twenty two or more data. The academic community has also developed its own tools. These have tended to replace the manufacturer's own analysis software. Such tools also use outlier detection and robust averaging. Some, such as GCRMA [55], ignore the control member of each pair.

Miller *et al.* separately normalised the natural log of the HG-U133A and HG-U133B values and then used MAS5 to calculate 44 928 gene expression values for each pair patient [37]. (Normalisation is needed to avoid the need to carefully control the amount of mRNA used and since Biologists are usually more interested in the relative strengths of gene activity, rather than absolute values.) Between 125 and 5 000 of the most variable were selected for further analysis. They used diagonal linear discriminant analysis to fit the *whole* data set. They say DLDA gave better results than k nearest neighbours and support vector machines. The DLDA p53 classifier used 32 genes.

Recently we have surveyed defects in more than ten thousand Affymetrix Gene Chips using a new technique [25, 28]. While [37] claims GeneChips with "visible artefacts" were re-run, we found spatial flaws in all their data. GeneChips should have an almost random speckled pattern due to the pseudo random placement of gene probes. The large light gray areas in Figure 3 indicate spatial flaws. Spatial flaws occur most often towards the edges of GeneChips. Figures 4 and 5 shows

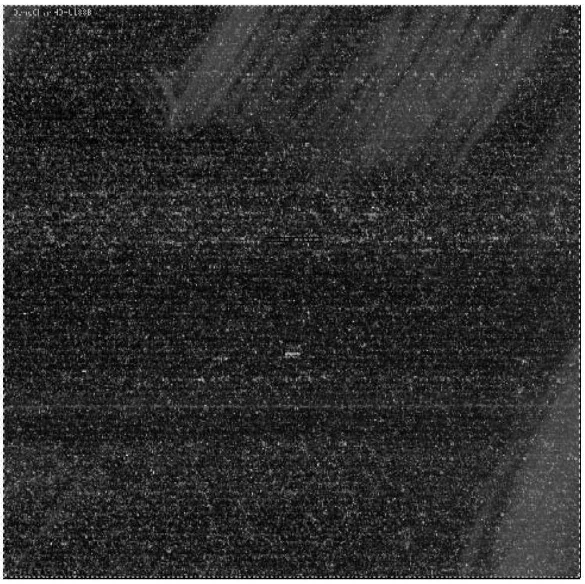


Fig. 3. First HG-U133B Breast Cancer GeneChip. Data have been quantile normalised. (This is like converting to a standard score and effectively replaces data by its logged value). Large spatial flaws can be seen at the top and lower right hand corners.

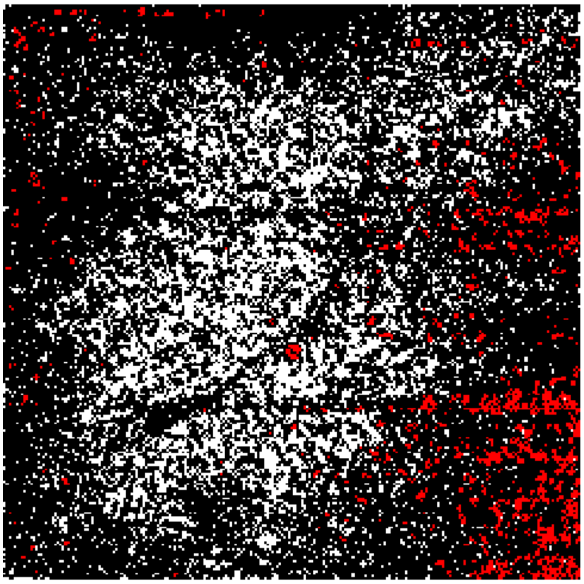


Fig. 4. Density of spatial flaws in 98 HG-U133A Breast Cancer GeneChips. Red more than twenty of 98 GeneChips are flawed (Black at least one).

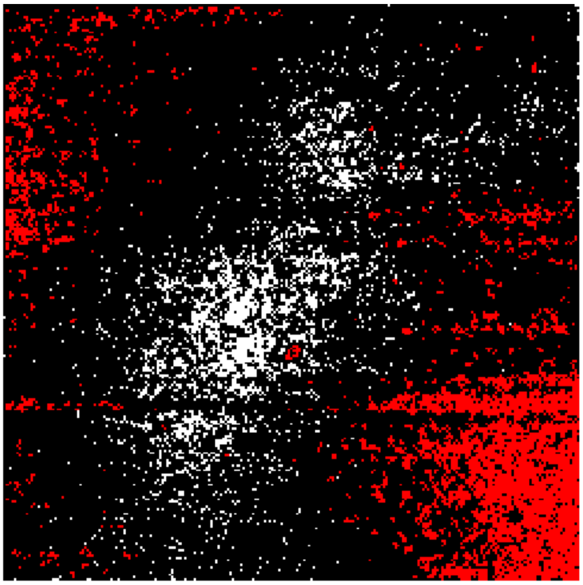


Fig. 5. Density of spatial flaws in 98 HG-U133B Breast Cancer GeneChips showing HG-U133B have more spatial errors than HG-U133A, c.f. Fig. 4.

the location and density of known errors in some data used for training GP and subsequent testing.

5 GeneChip Data Mining Using Genetic Programming on a GPU

Section 3 has listed the previous experiments evolving programs with a GPU. These have either represented the programs as trees or as networks (Cartesian GP) [16] and used the GPU for fitness evaluation. Harding compiled his networks into GPU programs before transferring the compiled code onto the GPU. We retain the traditional tree based GP and use an interpreter running on the GPU. Next we shall briefly recap how to interpret multiple programs simultaneously on a SIMD computer [20] and then detail tricks needed to address 512MBytes on a GPU.

Essentially the interpreter trick is to recognise that in the SIMD model the "single instruction" belongs to the interpreter and the "multiple data" are the multiple GP trees. The single interpreter is used by millions of programs. It is quite small and needs to be compiled only once. It is loaded onto every stream processor within the GPU. Thus every clock tick, the GPU can interpret a part of 128 different GP trees. The guts of a standard interpreter is traditionally a n-way switch where each case statement executes a different GP opcode. A SIMD machine cannot (in principle) execute multiple different operations at the same time. However they do provide a `cond` statement.

A `cond` statement has three arguments. The first is the control. It decides which of the other two arguments is actually used. `cond` behaves as if the calculations needed by its second and third arguments are both performed, but only one is used. Which one depends upon the `cond`'s first argument.

We use conditional statements like `x=cond(opcode=='+', a+b, x)` to perform an operation only if required. If the current instruction is `+` `cond` sets `x` to `a+b`. Otherwise it does nothing (by setting `x` to itself). (See Figures 6, 7 and 8.) Note the SIMD interpreter executes every `cond` for every instruction in the program. (In a normal interpreter a switch statement would direct the interpreter to execute just the code needed for the current instruction.) If there are five opcodes, this means for every leaf and every function in the program, the opcode at that point in the tree will be obeyed once but so too will four `cond` no-ops. As we showed in [26] the no-ops and indeed the functions cost almost nothing. It is reading the inputs from the training data which is expensive.

GPUs, at present, cannot imagine anyone having a screen bigger than 2048×2048 and therefore do not support arrays with more than 2^{22} elements. Each training example has data from both HG-U133A and HG-U133B, i.e. $2 \times 712^2 = 1\,013\,888$ floats. Therefore we pack four training examples per array. Since we split the available data into more or less equal training and holdout sets, the GPU fitness evaluation code need process at most only half the 251 patients' data at a time. The code allows 32 arrays (i.e. upto 128 patients). This occupies 512MB. All data transfers and data conversions are performed automatically by RapidMind's package. Rapid-

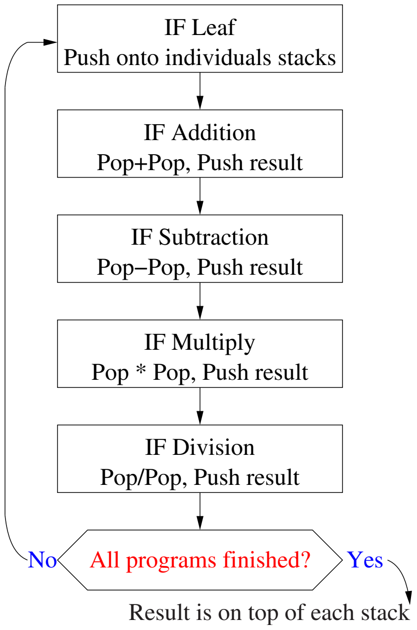


Fig. 6. The SIMD interpreter loops continuously through the whole genetic programming terminal and function sets for everyone in the population. GP individuals select which operations they want as they go past and apply them to their own data and their own stacks.

```
#define OPCODE(PC) ::PROG[PC+(prog0*LEN)]
PC=0;
FOR(PC,PC<(LEN-1),PC++) {
  //if leaf push data onto stack
  top = cond(OPCODE(PC)=='+', stack(1)+stack(0), top);
  top = cond(OPCODE(PC)=='-', stack(1)-stack(0), top);
  top = cond(OPCODE(PC)=='*', stack(1)*stack(0), top);
  top = cond(OPCODE(PC)=='/', stack(1)/stack(0), top);
  //remaining stack operation not shown
 } ENDFOR
```

Fig. 7. GPU Reverse Polish Notation SIMD interpreter. `prog0` indicates which RPN program is being evaluated on which stream processor. The central loop cycles through all operations on all stream processors. Each individual program uses `cond` statements to execute only those operations it needs.

Mind keeps track of when data are used and modified. Since the training data are not modified, they are stored in the GPU at the start of the run. Each generation, only the data which has changed, i.e. the GP individuals and their fitness's, are transfered between the host computer and the GPU. The architecture is shown shown in Figure 9.

```
Value<8,float> stack;

#define PUSH(V) \
join(join(V,stack(0,1,2)),stack(3,4,5,6))

//conditionally POP stack (fake by using rotation)
#define OP3(XCODE,OP) \
stack = cond(XCODE==OPCODE, \
             join(OP,stack(2,3,4),stack(5,6,7,1)), \
             stack);
```

Fig. 8. Partial implementation for GPU stack operations. Since RapidMind does not support index operations on writing to arrays the whole stack is updated. On PUSH the eight element stack is shuffled to the left using nested `join()` and the value is placed in `stack(0)`. The upper most element is lost. GP genetic operations ensure tree depth does not exceed eight and so there can be no stack overflow. (However GP can evolve solutions which happily cause stack over run. Nature will find a way.) `OP3` uses `cond` so that the operation `OP` on the two elements on top of the stack only takes place if the current instruction `OPCODE` is the right one. Then the stack is shifted down one place and the result of the operation is put in `stack(0)`.

The interpreter has to be structured to work within another GPU restriction. Like most other GPUs, the nVidia 8800 allows each GPU program at most sixteen inputs. I.e. the interpreter cannot access all 32 training data arrays simultaneously. Since it must access other data arrays (programs, fitness, debugging, etc.) as well as the training arrays, the interpreter was split into four equal parts, each of which deals with eight arrays (i.e. upto 32 patients). A parameterised C++ macro is used to define the interpreter code for one array. To access the 32 arrays of training data, the macro is used eight times in each of the four programs.

The four sets of outputs are summed and combined into a single fitness value per GP individual. For convenience the summation and fitness calculation are done by three auxiliary GPU programs. Only the final result is transfered to the host computer. RapidMind's optimising compiler deals with all seven GPU programs as one unit and therefore can, in principle, optimise across their boundaries. C++ code to invoke the GPU via RapidMind is shown in Figure 10.

As described in [29] the interpreter represents the GP trees as linearised reverse polish expressions. By using a stack these can be evaluated in a single pass. For simplicity, the expressions are all the same length. Smaller trees are simply padded with no-ops. Because of the enormous number of inputs, it is no longer possible to code each opcode into a byte [29] instead at least 20 bits are needed. In fact we use a full word per opcode. This means a population of five million fifteen node programs can be stored in 320Mbyte on the PC. Here we again run into the 2^{22} GPU addressing limit. Since each program occupies sixteen words (fifteen, plus one for a stop code), the population is broken into twenty 256k units.

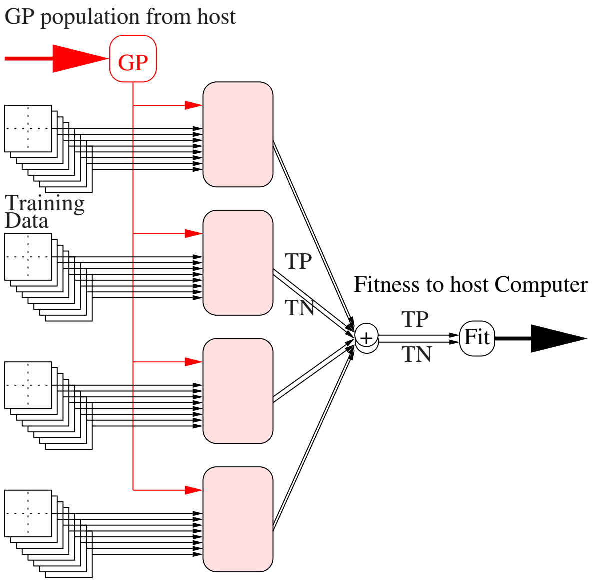


Fig. 9. GPU software architecture needed to overcome 2^{22} and no more than sixteen arrays GPU limits in order to access 512MB of training data and a population of five million GP programs. The population is split into twenty 256k parts by the host CPU. Twenty times per generation 256 thousand GP programs are passed to GPU (red) and interpreted by it. On average, the GPU takes slightly less than a second to interpret them and return their fitness values. There are four parameterised instances of the SIMD interpreter (pink). Each deals with upto 32 training cases. Each uses 1+8+2 arrays (plus others for control and debug, not shown, total 12 or more). Each instance is limited to sixteen arrays. We pack four sets of patient data (4×1013888) per array. Four groups of eight arrays allows 512M of training data. After running each group of $\frac{1}{4}$ million programs, $\frac{1}{4}$ million fitness values are returned to the host PC.

It takes slightly less than a second to evaluate all 262 144 programs. This fits tolerably well with our earlier finding [29] that, to get the best from the GPU, its work should be fed into the GPU in units of between 1 and 10 seconds.

5.1 GP for Large Scale Data Mining

We have previously described using genetic programming to data mine GeneChip data [24]. Our intention is to automatically evolve a simple (possibly non-linear) classifier which uses few simple inputs to predict the future about ten years ahead. To ensure the solutions are simple (and for speed) the GP trees are limited to fifteen nodes. (Whilst this is obviously small, it is not unreasonable. For example, Yu *et al.* successfully evolved classifiers limited to only eight nodes [56].)

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace std;
using namespace rapidmind;

const int NP = 2560*2048; //NP is Number of programs in Population
const int LEN =15+1;      //Max GP individual length, allow stop code

//Number gp individual Programs loaded onto GPU
const int GPU_NP = 4*1024*1024/LEN; //22bit limit

//virtual array prog0 is used to simulate indexOf
Array<1,Value1i> prog0 = grid(GPU_NP);

for(int n=0;n<(NP/GPU_NP);n++) {
  // Access the internal arrays where the data is stored
  unsigned int* in_PROG = PROG.write_data();
  memcpy(in_PROG,&Pop[n*GPU_NP*LEN],LEN*GPU_NP*opsize);

  Array<1,Value1i> TP0;
  Array<1,Value1i> TN0;
  Array<1,Value1i> TP1;
  Array<1,Value1i> TN1;
  Array<1,Value1i> TP2;
  Array<1,Value1i> TN2;
  Array<1,Value1i> TP3;
  Array<1,Value1i> TN3;
  Array<1,Value1i> TP;
  Array<1,Value1i> TN;

  Array<1,Value1f> F;

  bundle(TP0,TN0) = gpu->m_update0(prog0);
  bundle(TP1,TN1) = gpu->m_update1(prog0);
  bundle(TP2,TN2) = gpu->m_update2(prog0);
  bundle(TP3,TN3) = gpu->m_update3(prog0);
  TP              = gpu->sum(TP0,TP1,TP2,TP3);
  TN              = gpu->sum(TN0,TN1,TN2,TN3);
  F               = gpu->fitness(TP,TN);

  const float* fit = F.read_data();
  memcpy(&output[n*GPU_NP],fit,GPU_NP*sizeof(float));
}//endfor each GPU sized element of Pop
```

Fig. 10. Part of C++ code to run GP interpreter on the GPU twenty times (`NP/GPU_NP`) per generation. At the start of the loop the next fragment of `Pop` is copied into RapidMind variable `PROG`. `PROG`'s address is given by `write_data()`. RapidMind variables `TP0` to `TN` are used to calculate fitness, cf. Figure 9. They are not used by the host CPU and are never transfered from the GPU to the CPU. The four `m_update*(prog0)` programs each run the GP interpreter on 256k programs on 32 patients' data. They are identical, except they are parameterised to run on different quarters of 128 training cases. The RapidMind `bundle()` provides a way that is compatible with C++ syntax for a GPU program to return two or more values. All evaluation is run on the GPU until `read_data()` is called. `read_data()` not only transfers the fitness values, in `F`, but also resynchronises the GPU and CPU.

In our earlier work we had only one GeneChip for each of the 60 patients (and that was an older design). Also the data set did not include the probe values but only 7129 gene expression values [24]. We now have the raw probe values (and compute power to use them). Therefore we will ask GP to evolve combinations of the probe values rather than use Affymetrix or other human designed combinations of them. This gives us more than a million inputs. The first step is to use GP as its own feature selector.

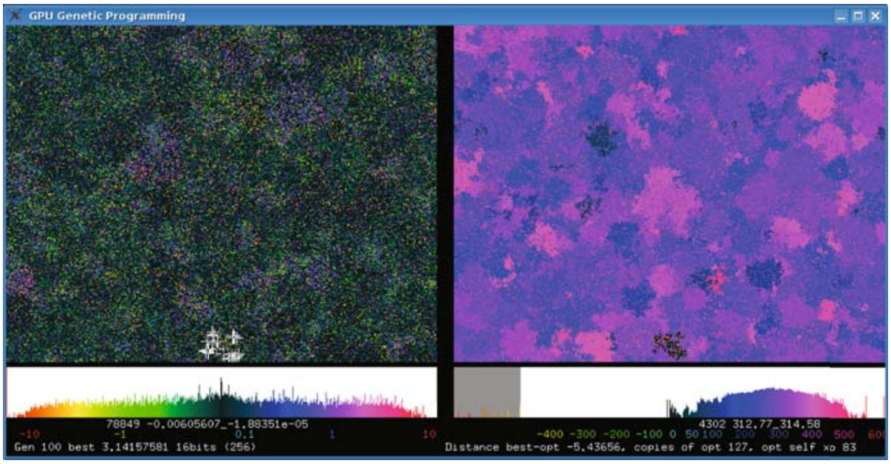
Essentially the idea is to use Price's theorem [46]. Price showed the number of fit genes in the population will increase each generation and the number of unfit genes will decrease. We run GP several times. We ignore the performance of the best of run individual and instead look at the genes it contains. The intention was the first pass would start with a million inputs and we would select in the region of 10 000 for the second pass. Then we would select about 100 from it for the third pass. Finally a GP run would be started with a much enriched terminal set containing only inputs which had showed themselves to be highly fit in GP runs. However we found only two selection passes were needed, cf. Section 6.

The question of how big to make the GP population can be solved by considering the coupon collector problem [10, p284]. On average $n(\log(n) + 0.37)$ random trials are needed to collect all of n coupons. Since we are using GP to filter inputs, we insist that the initial random population contains at least one copy of each input. That is we treat each input as a coupon (so $n = 1\,013\,888$) and ask how many randomly chosen inputs must we have in the initial random population to be reasonably confident that we have them all. The answer is 14 million. If we overshoot by a few thousands, we are sure to get all the leafs into the initial population. Since a program of fifteen nodes has eight leafs and half of these are constants we need at least $\frac{1}{4}(14 \text{ million}) = 3.6$ million random trees. An initial population of five million ensures this.

In [29] we used a fairly gentle selection pressure. Here we need our programs to compete, so the tournament size was increased to four. However we have to be cautious. At the end of the first pass, we want of the order of 100 000 inputs to chose from. This means we need about 25 000 good programs (each with about four inputs). We do not want to run our GP 25 000 times. The compromise was to use overlapping fine grained demes [19] to delay convergence of the population, cf. Figure 11. The GP population is laid out on a rectangular 2560×2048 grid (cf. Figure 12). This was divided into eighty 256×256 squares. At the end of the run, the genetic composition of the best individual in each square was recorded. Note to prevent the best of one square invading the next, parents were selected to be within 10 grid points of their offspring. Thus genes can travel at most 100 grid points in ten generations. The GP parameters are summarised in Table 1.

5.2 Data Sets

As part of our large survey of GeneChip flaws [28] we had already down loaded all the HG-U133A and HG-U133B data sets in GEO [3] (6685 and 1815 respectively)



```
78849 -0.00605607_-1.88351e-05
Gen 100 best 3.14157581 16bits (256)
4302 312.77_314.58
Distance best-opt -5.43656, copies of opt 127, opt self xo 83
```

Fig. 11. Screen shot of a 512 × 400 GP population, i.e. 204 800 programs (from run approximating π [29]) evolving under selection, crossover and subtree mutation after 100 generations. Colour indicates fitness (left) and syntax (right). Below are two histograms (log scale) showing distribution of population by fitness and genotypic distance from the first optimal solution. (Colour scales below each histograms.) Local convergence and the production of species is visible (esp. right). See http://www.cs.ucl.ac.uk/staff/W.Langdon/pi2_movie.html and Google videos for animation and more explanation.

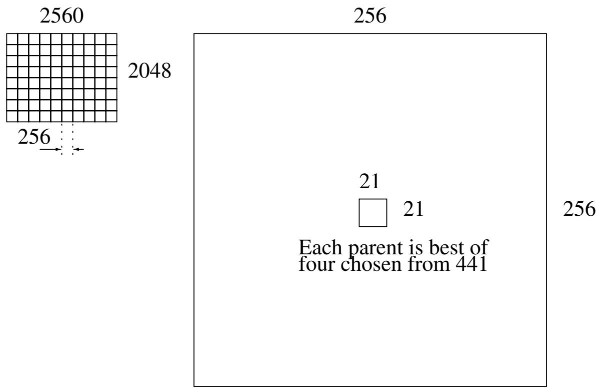


Fig. 12. Left: The GP population of five million programs is arranged on a 2560 × 2048 grid, which does not wrap around at the edges. At the end of the run the best in each 256 × 256 tile is recorded. Right: (note different scale) parents are drawn by 4-tournament selection from within a 21 × 21 region centred on their offspring.

Table 1. GP Parameters for Uppsala Breast Tumour Biopsy

Function set:	ADD SUB MUL DIV operating on floats
Terminal set:	712^2 Affymetrix HG-U133A and 712^2 HG-U133B probe mRNA concentrations. 1001 Constants -5, -4.99, -4.98, ... 4.98, 4.99, 5
Fitness:	AUROC $\left(\frac{1}{2}\frac{\text{TP}}{\text{No. pos}} + \frac{1}{2}\frac{\text{TN}}{\text{No. neg}}\right)$ less 1.0 if number of true positive cases (TP=0) or number of true negative cases (TN=0) [23].
Selection:	tournament size 4 in overlapping fine grained 21×21 demes [19], non elitist, Population size 2560×2048
Initial pop:	ramped half-and-half 1:3 (50% of terminals are constants)
Parameters:	50% subtree crossover. 50% mutation (point 22.5%, constants 22.5%, subtree 5%). Max tree size 15, no tree depth limit.
Termination:	10 generations

and calculated a robust average for each probe. These averages across all these human tissues were used to normalise the 251 pairs of HG-U133A and HG-U133B GeneChips and flag locations of spatial flaws. (Cf. Figures 3–5. R code to quantile normalise and detect spatial flaws is available via http://bioinformatics.essex.ac.uk/users/wlangdon.) The value presented to GP is the probe's normalised value minus its average value from GEO. This gives an approximately normal distribution centred at zero. Cf. Figure 13.

The GeneChip data created by [37] were obtained from NCBI's GEO (data set GSE3494). Other data, e.g. patients' age, survival time, if breast cancer caused death and tumour size, were also down loaded. Whilst [37] used the whole dataset: with

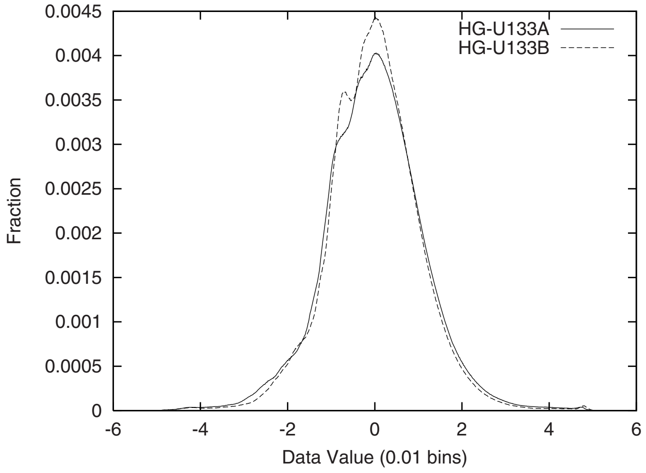


Fig. 13. Uppsala breast cancer distribution of log deviation from average value.

more than a million inputs we were keen to avoid over fitting, therefore the data were split into independent training and verification data sets.

Initially 120 GeneChip pairs were randomly chosen for training but results on the verification set were disappointing. Accordingly we redesigned our experiment to chose training data in a more controlled fashion. To reduced scope for ambiguity we excluded patients who: a) survived for more than 6 years before dying of breast cancer, b) survived for less than 9.8 years before dying of some other cause, c) patients where the outcome was not known. We split the remaining data as evenly as possible into training (91) and verification (90) sets.

It is known that age plays a prominent role in disease outcomes but the patients were from 28 to 83 years old. So we ordered the data to ensure both datasets had the same age profile. We also balanced as evenly as possible outcome (140 v. 41), tumour size, estrogen receptor (ER) status and progesterone receptor (PgR) status.

6 Results

GP was run one hundred times with all inputs taken from the 91 training examples using the parameters given in Table 1. After ten generations the best program in each of the eighty 256 × 256 squares was recorded. The distribution of inputs used by these 100 × 80 programs is given in Figure 14. Most probes were not used by any of the 8000 programs. 24 810 were used by only one. 2091 by two, and so on.

The 3422 probes which appeared in more than one of the 8000 best of generation ten programs were used in a second pass. In the second pass GP was also run 100 times.

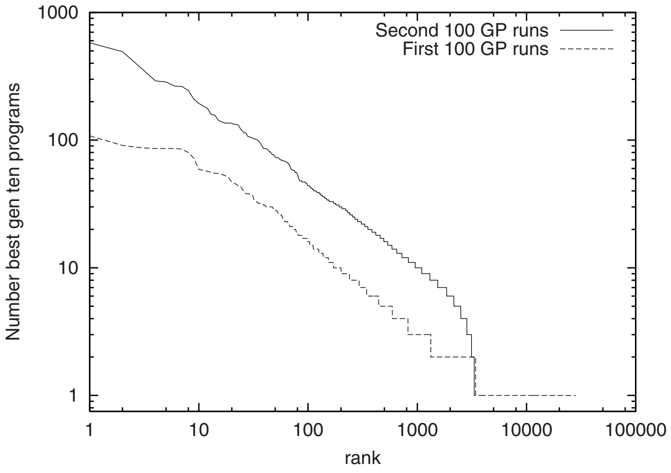


Fig. 14. Distribution of usage of Affymetrix probe in 8000 best of generation 10 GP programs. Both distributions are almost a straight lines (note log scales). Cf. Zipf's law [57].

Eight probes appeared in more than 240 of the best 8000 programs of the second pass. These were the inputs to a final GP run. (The GP parameters were again kept the same).

The GP found several good matches to the 91 training examples. Ever mindful of overfitting [6], as a solution we chose one with the fewest inputs (three). GP found a non-linear combination of two PM probes and one MM probe from near the middle of HG-U133A, cf. Figure 15 and Table 2. The evolved predictor is the sum of two non-linear combination of two genes (decorin/C17orf81 and C17orf81($2.94 + 1/$S-adenosylhomocysteine hydrolase), cf. Figure 16). Both sub-expressions have some predictive ability. The three probes chosen by GP are each highly correlated with all PM probes in their probeset and so can be taken as a true indication of the corresponding gene's activity. The gene names where given by the manufacturer's netaffx www pages. Possibly terms like decorin/C17orf81 are simply using division as a convenient way to compare two probe values. Indeed the sign indicates if two values are both above or both below average.

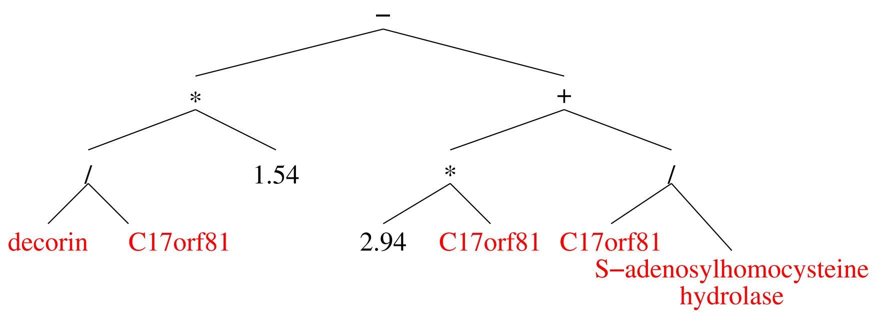


Fig. 15. GP evolved three input classifier. (Using Affymetrix probe names) survival is predicted if $1.54\frac{\text{201893\_x\_at.2pm}}{\text{219260\_s\_at.7pm}} - 2.94\,\text{219260\_s\_at.7pm} - \frac{\text{219260\_s\_at.7pm}}{\text{200903\_s\_at.8mm}} < 0$.

The evolved classifier gets 70% of the verification set correct. If we use the three input predictor on the whole Uppsala dataset (excluding the fifteen cases where the outcome is not known), it gets right 184 out of 236 (78%). Figure 17 shows this non-linear classifier gives a bigger separation between the two outcomes than a 32-gene model requiring non-linear calculation of more than seven hundred probe values [37, Fig. 3 B].

We tried applying our evolved classifier to a different Breast tumour dataset [44]. Unfortunately we have less background data and no details of follow up treatment for the second group of patients. Also they were treated in another hospital a decade later. Undoubtedly cancer treatment has changed since our data was collected. These, and other differences between the cohorts, may have contributed to the fact that our classifier did less well on the second patient cohort. For example, the Kaplan survival plot to eight years [25, Figure 6] is less well separated than in Figure 17 for twelve years.

Table 2. Top twenty Affymetrix probes used most in 8000 best of generation 10 second pass GP programs. Cf. Figure 14. The top eight were used in the final GP run.

	Used	X,Y	chiptype	Affy id		NetAffx Gene Title
1	579	350,514	A	200903_s_at	8.mm	S-adenosylhomocysteine hydrolase
2	493	325,511	A	219260_s_at	7.pm	C17orf81. chromosome 17 open reading frame 81
3	363	254,667	A	201893_x_at	2.pm	decorin
4	291	392,213	A	219778_at	4.pm	zinc finger protein, multitype 2
5	286	366,310	B	230984_s_at	10.mm	230984_s_at was annotated using the Accession mapped clusters based pipeline to a UniGene identifier using 17 transcript(s). *This assignment is strictly based on mapping accession IDs from the original UniGene design cluster to the latest UniGene design cluster*.
6	265	324,484	A	216593_s_at	9.mm	phosphatidylinositol glycan anchor biosynthesis, class C
7	263	542,192	B	233989_at	4.mm	EST from clone 35214, full insert. UniGene ID Build 201 (01 Mar 2007) Hs.594768 NCBI
8	245	269,553	B	223818_s_at	2.pm	remodeling and spacing factor 1
9	209	416,107	B	226884_at	10.pm	leucine rich repeat neuronal 1
10	194	613,230	B	235262_at	6.mm	Zinc finger protein 585B. 235262_at was annotated using the Accession mapped clusters based pipeline to a UniGene identifier using 7 transcript(s). *This assignment is strictly based on mapping accession IDs from the original UniGene design cluster to the latest UniGene design cluster.*
11	185	61,573	A	221773_at	4.pm	ELK3, ETS-domain protein (SRF accessory protein 2)
12	177	619,316	B	235891_at	6.mm	235891_at was annotated using the Genome Target Overlap based pipeline to a UCSC Genes,ENSEMBL ncRNA identifier using 2 transcript(s).
13	159	531,613	A	NA		
14	157	426,349	A	213706_at	11.pm	glycerol-3-phosphate dehydrogenase 1 (soluble)
15	144	57,434	B	242689_at	10.mm	Ral GEF with PH domain and SH3 binding motif 1. 242689_at was annotated using the Accession mapped clusters based pipeline to a UniGene identifier using 5 transcript(s). *This assignment is strictly based on mapping accession IDs from the original UniGene design cluster to the latest UniGene design cluster.*
16	140	15,353	A	213071_at	4.pm	dermatopontin
17	137	65,606	B	229198_at	6.mm	ubiquitin specific peptidase 35
18	136	107,597	A	202995_s_at	4.pm	fibulin 1
19	136	108,393	A	209615_s_at	5.pm	p21/Cdc42/Rac1-activated kinase 1 (STE20 homolog, yeast)
20	136	135,279	A	202995_s_at	2.pm	fibulin 1

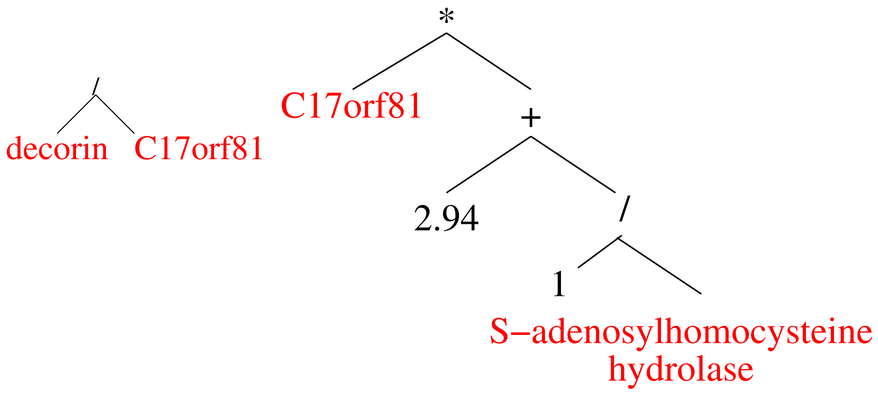


Fig. 16. The GP classifier (Figure 15) is the weighted addition of two two input classifiers (left and right).

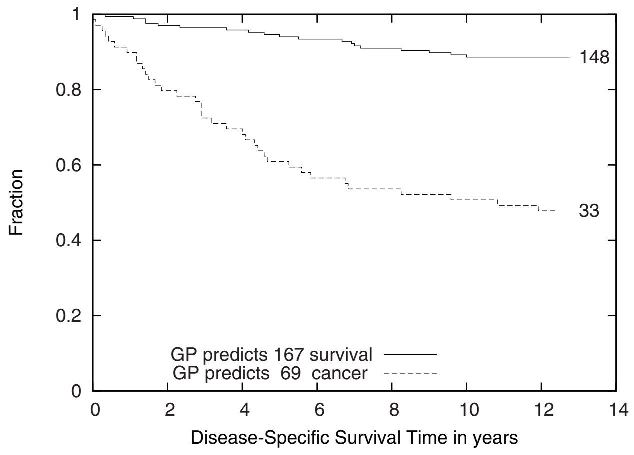


Fig. 17. Kaplan-Meier survival plots, such as the one above, are often used to measure the fraction of patients surviving a certain time after treatment (in this case breast cancer surgery). The three input GP classifier (given in Figure 15) predicts 167 survivors and 69 breast cancer fatalities. The right end of the top line shows that 148 of the 167 predicted to survive lived for more than 12 years. In contrast the lower curve refers to the 67 patients whose gene expression values suggested they would not survive ten years (However 33 of the 67 lived at least $12\frac{1}{2}$ more years).

7 Discussion/Practicalities

In [26] we present detailed timing arguments which show the GP RapidMind interpreter is limited not by the calculations need to interpret the millions of programs but the time taken to fetch their inputs from the GPU's own memory, cf. Figure 18.

So replacing interpreted code by compiled code, without addressing the memory bottle neck, would give negligible speed up. Indeed the interpreter is already faster than some compiled GPU approaches.

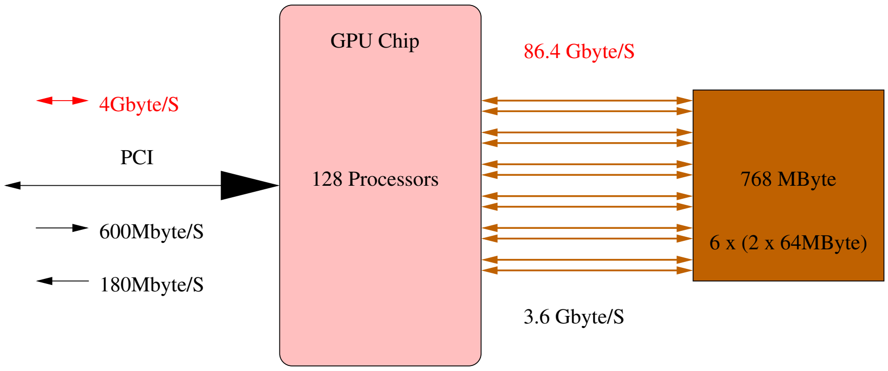


Fig. 18. nVidia 8800 Block diagram. The 128 Stream Processors are connected to the host PC via its PCI express bus. Measurements show RapidMind data both into the GPU and back to the host are efficient (600 and 180 Megabytes per second, i.e. about three quarters of the maximum possible with this PCI). 4 Gigabytes per second is likely to be available soon. The 8800 has twelve 64 megabyte RAM chips. These are paired to give six 16 million $\times$ 64 bit words of storage. Each is connected to the GPU main silicon die by its own 64 bit wide bus. In principle this gives 86.4 GBytes per second of on board memory I/O, however in practise with RapidMind it is impossible to use more than one the six buses simultaneously. Nevertheless it appears that multi-threading of 32-bit access enables the 128 stream processors to obtain about 3.6 GBytes per second.

To a first approximation, any artificial intelligence supervised learning technique, which used this training data in the same way will take about a second or more to test $\frac{1}{4}$ million random classifiers; be they rules, artificial neural networks or programs.

7.1 Speed Up

For this application, the GP interpreter's runs 535 million GP operations per second. 535 MGPop/S is only slightly less than we measured previously [29] with training sets containing ten times as many examples but only about 5kB of training data in total.

To determine speed up, the RapidMind C++ GPU interpreter was converted into a normal C++ GP interpreter and run on the same CPU as was used to host the GPU. I.e. an Intel CPU 6600 2.40GHz. Within the differences of floating point rounding, the GPU program and the new program produced the same answers but in terms of the fitness evaluation the GPU ran 7.59 times faster.

On a different example with more training examples but each containing much less data we obtained a GPU speed up of 12.6 [29]. The GPU interpreter's performance on a number of problems has been in the region $\frac{1}{2}$ to 1 giga GPops, cf. Table 3. In contrast the performance of compiled GPs on GPUs has varied widely, e.g. with number of training examples and program size.

Table 3. Nvidia GeForce 8800 GTX. Genetic Programming Primitives Interpreted Per Second.

Experiment	No. of Terminals Inputs+Consts	Functions	Population size	Program size	Stack depth	Test cases	Speed 10^6 OP/S
Mackey-Glass	8+128	4	204 800	11.0	4	1200	895
Mackey-Glass	8+128	4	204 800	13.0	4	1200	1056
Protein	20+128	4	1 048 576	56.9	8	200	504
Laser_a	3+128	4	18 225	55.4	8	151 360	656
Laser_b	9+128	4	5 000	49.6	8	376 640	190
Cancer	1 013 888+1001	4	5 242 880	$\leq$15.0	4	128	535
GeneChip	47+1001	6	16 384	$\leq$63.0	8	200[a]	314
Sextic[b]	1+na	8	12 500	66.0	17	1024	650[c]

[a] The 200 test cases used were randomly sampled from 300 000 available every generation
[b] $x^6 - 2x^4 + x^2$ approximated by a CUDA system [48] using an optimised RPN interpreter
[c] If we excluded Java code running on the host PC and considered only fitness evaluation on the GPU 1300 MGPop/S was achieved.

7.2 Computational Cube

In genetic programming fitness evaluation, which usually totally dominates run time, can be thought of along three dimensions: 1) the population 2) the training examples and 3) the programs or trees themselves. While it need not be the case, often the GP uses a generational population. Meaning:

1. the whole population is evaluated as a unit before the next generation is created.
2. Often either the whole of the training data, or the same subset of it, is used to calculate the fitness of every member of the population. (Sometimes, in other work, between generations we change which subset is in use.)
3. In many, but by no means all, cases the programs to be tested have a maximum size and do not contain dynamic branches, loops, recursion or function calls. Even for trees, this means the programs can be interpreted in a single pass through a maximum number of instructions. (Shorter programs could, in principle, be padded with null operations.)

We can think of these three dimensions as forming a cube of computations to be done. See Figure 19.

In our implementation (Section 5) the computational cube is sliced vertically (Figure 19) with one GPU thread for each program and each thread looking after all the fitness cases for an individual program. Explicit code in the thread works along

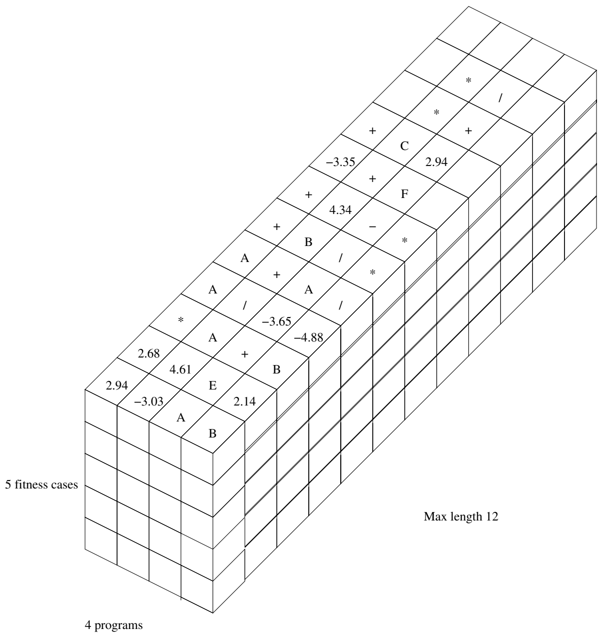


Fig. 19. Evaluating a GP population of four individuals each on the same five fitness cases. There are upto $4 \times 5 \times 12$ GP operations to be performed by, in principle, 240 GPU threads. Each cube needs the opcode to be interpreted, the fitness test case (program inputs) and the previous state of the program (i.e. the stack).

the length of the program and processes all the fitness cases for that program. We believe this model of parallel processing works well generally.

Recently we have implemented horizontal slicing. That is, each fitness case has its own GPU thread. The fundamental switch in the GP interpreter makes little difference to the GPU and is readily implemented. Indeed in this respect the GPU is quite flexible. It is relatively straightforward to radically re-arrange the way in which the GPU parallel hardware is used. We have not as yet tried slicing the computational cube along the programs' lengths.

In principle it is possible for each GP instruction to be executed in a different computational thread. In normal programs this would not be contemplated since the complete computational state would have to be passed through each thread. However the complete state for many GP applications is purely the stack. In many cases this is quite small. Therefore executing each function and each GP terminal in a separate GPU thread could be considered. This dimension, also requires dealing with programs that are of different lengths. It is also unattractive since variable data needs to be passed, whilst the corresponding data along the other dimensions are not modified, which saves writing them back to memory.

The efficient use of current GPUs requires many active threads, perhaps upto sixty four per stream processor. With a powerful GPU this means thousands of threads must be kept active to get the best from the hardware, cf. Figure 20. While the computational cube is an attractive idea it is easy to see that far from having too few threads it would be easy to try to divide a GP fitness computation into literally millions of parallel operations, which could not be efficiently implemented. However dividing it along two of the possible three planes is effective.

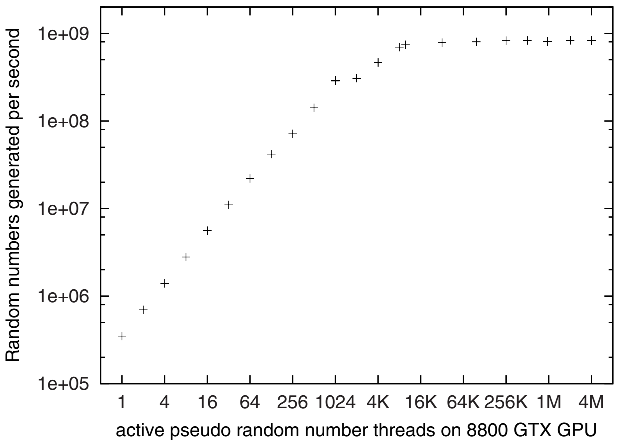


Fig. 20. Park-Miller random numbers per second (excluding host-GPU transfer time) on nVidia 8800 GTX. In the test environment the rate depends upon how effectively the 128 parallel stream processors can be used. Only when there are more than 8192 separate threads do the 128 stream processors effectively saturate [22].

7.3 Tesla and the Future of General Purpose GPU Computing

Unsurprisingly a large fraction of the 618 10^6 transistors of the GPU chip are devoted to graphics operations, such as anti-aliasing. This hardware in unlikely to be useful for scientific computing and so represents an overhead. It appears the newly introduced Tesla cards retain this overhead. However if Tesla makes money, the next generation of GPGPU may trade transistors to support graphics operations for transistors to support more scientific data manipulation. E.g. for bigger on chip caches.

7.4 Absence of Debugger and Performance Monitoring Tools

RapidMind allows C++ code to be moved between the CPU, the GPU and CELL processors without recompilation. Their intention is the programmer should debug C++ code on the CPU. This allows programmers to use their favourite programming environment (IDE), including compiler and debug tools. Recently RapidMind has introduced a "debug backend" but it too actually runs the code being debugged on the host CPU. Linux GNU GCC/GDB and Microsoft visual C++ are both supported. Owens *et al.* say Google's PeakStream, which has some similarities with RapidMind but was inspired by Brook, "is the first platform to provide profiling and debugging support" [42, p886].

The RapidMind performance log can be configured to include details about communication between the CPU and the GPU. Details include, each transfer, size of transfer, automatic data conversion (e.g. unsigned byte to GPU float) and representation used on the GPU. (E.g. texture size, shape and data type.) However for the internal details of GPU performance and location of bottle necks, one is forced to try and infer them by treating the GPU as a black box.

Recent software advances under the umbrella term of general purpose computing on GPUs (GPGPU) have considerably enhanced the use of GPUs. Nevertheless, GPU programming tools for scientific and/or engineering applications are primitive and getting the best out of GPUs "remains something of a black art" [42, p896,p897]. This is exacerbated by 1) the small number and consequent instability of hardware and software vendors in the GPGPU market. 2) Hardware specific program interfaces (APIs) which have been much more likely to require modification to existing programs to take advantage of new hardware than the corresponding interfaces in CPUs. 3) Lack of vendor independent APIs [42].

For GPU manufactures GPGPU remains an add-on to their principal market: games. Accompanying the rapid development in hardware they make corresponding changes in their software. This means the manufacturer's APIs tend to tested and optimised for a few leading games. This can have unfortunate knock effects on GPGPU applications [42]. Potentially GPU developers can isolates themselves from this by using higher level tools or languages, like RapidMind.

Despite their undoubted speed, if GPUs remain difficult to use, they will remained limited to specialised niches. To quote John Owens "Its the software, stupid" [41].

7.5 *C++ Source Code*

C++ code can be down loaded via anonymous ftp or `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/gpu_gp_2.tar.gz` Also `gpu_gp_1.tar.gz` has a small introductory example [29]. Whereas `random-numbers/gpu_park-miller.tar.gz` is for generating random numbers [22].

8 Conclusions

We have taken a large GeneChip breast cancer biopsy dataset with more than a million inputs to demonstrate a successful computational intelligence application running in parallel on GPU mass market gaming hardware (an nVidia GeForce 8800 GTS). We find a 7.6 speed up.

Initial analysis of the GPU suggests that the major limit is access to its 768Mbytes where the training data is stored. Indicating that, if other computational intelligence techniques, access the training data in similar ways, they would suffer the same bottle neck.

Whilst primarily interested in mutation of the p53 gene, Miller *et al.* tried support vector machines and k nearest neighbour but say diagonal linear discriminant analysis worked better for them [37]. They used DLDA to construct a non-linear model with more than 704 data items per patient. The non-linear model evolved by genetic programming uses only three. It has been demonstrated on a separated verification dataset. As Figure 17 shows, on all the available labelled data (236 cases), the classifier evolved using a GPU gives a wider separation in the survival data.

Acknowledgements

I would like to thank Denis Robilliard, Derek Foster, John Owens, Lance Miller, Simon Harding, Nolan White, Paul Price, Joanna Armbruster and Nick Holby.

References

1. Banzhaf, W., Harding, S., Langdon, W.B., Wilson, G.: Accelerating genetic programming through graphics processing units. In: Genetic Programming Theory and Practice VI, May 15-17, ch. 15. Springer, Ann Arbor (2008)
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction. Morgan Kaufmann, San Francisco (1998)
3. Barrett, T., Troup, D.B., Wilhite, S.E., Ledoux, P., Rudnev, D., Evangelista, C., Kim, I.F., Soboleva, A., Tomashevsky, M., Edgar, R.: NCBI GEO: mining tens of millions of expression profiles–database and tools update. Nucleic Acids Research 35(Database issue), D760–D765 (2007)
4. Charalambous, M., Trancoso, P., Stamatakis, A.: Initial experiences porting a bioinformatics application to a graphics processor. In: Bozanis, P., Houstis, E.N. (eds.) PCI 2005. LNCS, vol. 3746, pp. 415–425. Springer, Heidelberg (2005)
5. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: Thierens, D., et al. (eds.) GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation, London, July 7-11, vol. 2, pp. 1566–1573. ACM Press, New York (2007)
6. Corney, D.P.A.: Intelligent Analysis of Small Data Sets for Food Design. PhD thesis, University College, London (2002)
7. Dowsey, A.W., Dunn, M.J., Yang, G.-Z.: Automated image alignment for 2D gel electrophoresis in a high-throughput proteomics pipeline. Bioinformatics 24(7), 950–957 (2008)
8. Ebner, M., Reinhardt, M., Albert, J.: Evolution of vertex and pixel shaders. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 261–270. Springer, Heidelberg (2005)
9. Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S.: GPU cluster for high performance computing. In: Proceedings of the ACM/IEEE SC2004 Conference Supercomputing (2004)

10. Feller, W.: An Introduction to Probability Theory and Its Applications, 2nd edn., vol. 1. John Wiley and Sons, Chichester (1957)
11. Fernando, R.: GPGPU: general general-purpose purpose computation on GPUs. NVIDIA Developer Technology Group. Slides (2004)
12. Fok, K.-L., Wong, T.-T., Wong, M.-L.: Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems 22(2), 69–78 (2007)
13. Gobron, S., Devillard, F., Heit, B.: Retina simulation using cellular automata and GPU programming. Machine Vision and Applications (2007)
14. Harding, S.L., Banzhaf, W.: Fast genetic programming and artificial developmental systems on GPUs. In: 21st International Symposium on High Performance Computing Systems and Applications (HPCS 2007), Canada, p. 2. IEEE Press, Los Alamitos (2007)
15. Harding, S.: Evolution of image filters on graphics processor units using Cartesian genetic programming. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence, Hong Kong, June 1-6, IEEE Press, Los Alamitos (2008)
16. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 90–101. Springer, Heidelberg (2007)
17. Harding, S.L., Miller, J.F., Banzhaf, W.: Self-modifying Cartesian genetic programming. In: Thierens, D., et al. (eds.) GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation, London, July 7-11, vol. 1, pp. 1021–1028. ACM Press, New York (2007)
18. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
19. Langdon, W.B.: Genetic Programming and Data Structures. Kluwer, Dordrecht (1998)
20. Langdon, W.B.: A SIMD interpreter for genetic programming on GPU graphics cards. Technical Report CSM-470, Department of Computer Science, University of Essex, Colchester, UK, July 3 (2007)
21. Langdon, W.B.: Evolving GeneChip correlation predictors on parallel graphics hardware. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence, Hong Kong, June 1-6, pp. 4152–4157. IEEE Press, Los Alamitos (2008)
22. Langdon, W.B.: A fast high quality pseudo random number generator for graphics processing units. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence, Hong Kong, June 1-6, pp. 459–465. IEEE Press, Los Alamitos (2008)
23. Langdon, W.B., Barrett, S.J.: Genetic programming in data mining for drug discovery. In: Ghosh, A., Jain, L.C. (eds.) Evolutionary Computing in Data Mining. Studies in Fuzziness and Soft Computing, ch. 10, vol. 163, pp. 211–235. Springer, Heidelberg (2004)
24. Langdon, W.B., Buxton, B.F.: Genetic programming for mining DNA chip data from cancer patients. Genetic Programming and Evolvable Machines 5(3), 251–257 (2004)
25. Langdon, W.B., da Silva Camargo, R., Harrison, A.P.: Spatial defects in 5896 HG-U133A GeneChips. In: Dopazo, J., Conesa, A., Al Shahrour, F., Montener, D. (eds.) Critical Assesment of Microarray Data, Valencia, December 13-14 (2007); Presented at EMERALD Workshop
26. Langdon, W.B., Harrison, A.P.: GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Computing 12(12), 1169–1183 (2008)
27. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer, Heidelberg (2002)
28. Langdon, W.B., Upton, G.J.G., da Silva Camargo, R., Harrison, A.P.: A survey of spatial defects in Homo Sapiens Affymetrix GeneChips. IEEE/ACM Transactions on Computational Biology and Bioinformatics (in press, 2009)

29. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 73–85. Springer, Heidelberg (2008)
30. Lindblad, F., Nordin, P., Wolff, K.: Evolving 3D model interpretation of images using graphics hardware. In: Fogel, D.B., et al. (eds.) Proceedings of the 2002 Congress on Evolutionary Computation, CEC 2002, pp. 225–230. IEEE Press, Los Alamitos (2002)
31. Liu, W., Schmidt, B., Voss, G., Schroder, A., Muller-Wittig, W.: Bio-sequence database scanning on a GPU. In: 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, April 25-29. IEEE Press, Los Alamitos (2006)
32. Liu, Y., De Suvranu: CUDA-based real time surgery simulation. Studies in Health Technology and Informatics 132, 260–262 (2008)
33. Loviscach, J., Meyer-Spradow, J.: Genetic programming of vertex shaders. In: Chover, M., Hagen, H., Tost, D. (eds.) Proceedings of EuroMedia 2003, pp. 29–31 (2003)
34. Luo, Z., Liu, H., Wu, X.: Artificial neural network computation on graphic process unit. In: Proceedings of the 2005 IEEE International Joint Conference on Neural Networks, IJCNN 2005, July-4 August 2005, vol. 1, pp. 622–626 (2005)
35. Manavski, S., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9(suppl. 2), S10 (2008)
36. Meyer-Spradow, J., Loviscach, J.: Evolutionary design of BRDFs. In: Chover, M., Hagen, H., Tost, D. (eds.) Eurographics 2003 Short Paper Proceedings, pp. 301–306 (2003)
37. Miller, L.D., Smeds, J., George, J., Vega, V.B., Vergara, L., Ploner, A., Pawitan, Y., Hall, P., Klaar, S., Liu, E.T., Bergh, J.: An expression signature for p53 status in human breast cancer predicts mutation status, transcriptional effects, and patient survival. Proceedings of the National Academy of Sciences 102(38), 13550–13555 (2005)
38. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38(8), 114–117 (1965)
39. NVIDIA GeForce 8800 GPU architecture overview. Technical Brief TB-02787-001_v0.9, Nvidia Corporation (November 2006)
40. NVIDIA CUDA compute unified device architecture, programming guide. Technical Report version 0.8, NVIDIA, February 12 (2007)
41. Owens, J.: Experiences with GPU computing. Presentation slides (2007)
42. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (2008); invited paper
43. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
44. Pawitan, Y., Bjohle, J., Amler, L., Borg, A.-L., Egyhazi, S., Hall, P., Han, X., Holmberg, L., Huang, F., Klaar, S., Liu, E.T., Miller, L., Nordgren, H., Ploner, A., Sandelin, K., Shaw, P.M., Smeds, J., Skoog, L., Wedren, S., Bergh, J.: Gene expression profiling spares early breast cancer patients from adjuvant therapy: derived and validated in two population-based cohorts. Breast Cancer Research 7, R953–R964 (2005)
45. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming (2008), http://lulu.com, http://www.gp-field-guide.org.uk (With contributions by J. R. Koza)
46. Price, G.R.: Selection and covariance. Nature 227, 520–521 (1970)
47. Reggia, J., Tagamets, M., Contreras-Vidal, J., Jacobs, D., Weems, S., Naqvi, W., Winder, R., Chabuk, T., Jung, J., Yang, C.: Development of a large-scale integrated neurocognitive architecture - part 2: Design and architecture. Technical Report TR-CS-4827, UMIACS-TR-2006-43, University of Maryland, USA (October 2006)

48. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Population parallel GP on the G80 GPU. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 98–109. Springer, Heidelberg (2008)
49. Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney, A.: High-throughput sequence alignment using graphics processing units. BMC Bioinformatics 8, 474 (2007)
50. Stamatakis, A.: RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. Bioinformatics 22(21), 2688–2690 (2006)
51. Upton, G.J.G., Cook, I.: Introducing Statistics, 2nd edn. Oxford University Press, Oxford (2001)
52. Wilson, G., Banzhaf, W.: Linear genetic programming GPGPU on Microsoft's Xbox 360. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence, Hong Kong, June1-6. IEEE Press, Los Alamitos (2008)
53. Wilson, G., Harding, S.: WCCI 2008 special session: Computational intelligence on consumer games and graphics hardware (CIGPU-2008). SIGEvolution 3(1), 19–21 (2008)
54. Wirawan, A., Kwoh, C., Hieu, N., Schmidt, B.: CBESW: sequence alignment on the PlayStation 3. BMC Bioinformatics 9(1), 377 (2008)
55. Wu, Z., Irizarry, R.A., Gentleman, R., Martinez-Murillo, F., Spencer, F.: A model-based background adjustment for oligonucleotide expression arrays. Journal of the American Statistical Association 99(468), 909–917 (2004)
56. Yu, J., Yu, J., Almal, A.A., Dhanasekaran, S.M., Ghosh, D., Worzel, W.P., Chinnaiyan, A.M.: Feature selection and molecular classification of cancer using genetic programming. Neoplasia 9(4), 292–303 (2007)
57. Zipf, G.K.: Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. Addison-Wesley Press Inc., Reading (1949)

A Review on Parallel Estimation of Distribution Algorithms

Alexander Mendiburu, Jose Miguel-Alonso, and Jose A. Lozano

Abstract. Estimation of Distribution Algorithms (EDAs) are a set of techniques that belong to the field of Evolutionary Computation. They are similar to Genetic Algorithms (GAs), in the sense that, given a problem, they use a population of individuals to represent solutions, and this population is made to evolve towards the most promising solutions. However, instead of using the usual GA-operators such as mutation or crossover, EDAs learn a probabilistic model that tries to capture the main characteristics of the problem. Based on this idea, several EDAs have been introduced in the last years, showing a good performance and being able to solve problems of different complexity. One important drawback of EDAs is the significant computational effort required by the utilization of probabilistic models, when applied to real-world problems. This fact has led the research community to apply parallel schemes to EDAs, as a viable way to reduce execution times. Schemes already proposed for GAs have been used as the foundation for these parallel schemes. In this chapter, we make a review of parallel EDAs, with a main focus: identifying those parts that are susceptible of parallelization. Then we describe a collection of parallelization strategies proposed in the literature. Additionally, we provide some recommendations for those that are considering the implementation of parallel EDAs on state-of-the-art parallel computers.

1 Introduction

The increasingly high computing power achievable from commodity computers has encouraged the design and implementation of non-trivial algorithms to solve different kinds of complex optimization problems. Some of these problems can be solved via an exhaustive search over the solution space, but in most cases this brute force

Alexander Mendiburu · Jose Miguel-Alonso · Jose A. Lozano
Intelligent Systems Group, The University of the Basque Country
e-mail: {alexander.mendiburu,j.miguel,ja.lozano}@ehu.es

approach is unaffordable. In these situations, heuristic methods (deterministic or non deterministic) are often used, which search inside the space of promising solutions. Some heuristic approaches are specifically designed to find good solutions for a particular problem, but others are presented as a general framework adaptable to many different situations.

Among this second group (general designs), there is a family of algorithms that has been widely used in the last decades: Evolutionary Algorithms (EAs). This family comprises, as main paradigms, Genetic Algorithms (GAs) [18, 25], Evolution Strategies [58], Evolutionary Programming [17] and Genetic Programming [28].

The main characteristic of these algorithms is that they use techniques inspired by the natural evolution of the species. In nature, species change across time; individuals evolve, adapting to the characteristics of the environment. This evolution leads to individuals with better characteristics. This idea can be translated to the world of computation, using similar concepts:

Individual: Represents a possible solution for the problem to be solved. Each individual has a set of characteristics (genes) and a fitness value (based on its genes) that denotes the quality of the solution it represents.

Population: In order to look for the best solution, a group of several individuals is managed. An initial population is created randomly, and will change across time, evolving towards members with different (and supposedly better) characteristics.

Breeding: Several operators can be used to emulate the breeding process present in nature: mixing different individuals (crossover) or changing a particular one (mutation). These operators are used to obtain new individuals, expected to be better than the previous ones.

In the last two decades, GAs have been widely used to solve different problems, improving in many cases the results obtained by previous approaches. However, GAs require a large number of parameters (for example, those that control the creation of new individuals) that need to be correctly tuned in order to obtain good results. Generally, only experienced users can do this correctly and, moreover, the task of selecting the best choice of values for all these parameters has been suggested to constitute itself an optimization problem [19]. In addition, GAs show a poor performance in some problems (deceptive and separable problems) in which the existing crossover and mutation operators do not guarantee that better individuals will be obtained changing or combining existing ones.

Some authors [25] have pointed out that making use of the relations between genes can be useful to drive a more “intelligent” search through the solution space. This concept, together with the limitations of GAs, motivated the creation of a new type of algorithms grouped under the name of Estimation of Distribution Algorithms (EDAs).

EDAs were introduced in the field of Evolutionary Computation in [42], although similar approaches can be previously found in [72]. In EDAs there are neither crossover nor mutation operators. Instead, the new population of individuals is sampled from a probability distribution, which is estimated from a database that contains the selected individuals from the current generation. Thus, the interrelations between the different variables that represent the individuals are explicitly expressed through the joint probability distribution associated with the individuals selected at each generation. A common pseudo-code for all EDAs is described in Fig. 1.

Pseudo-code for the EDA framework.

```
Step 1. Generate the first population $D_0$ of $M$ individuals and evaluate all of them
Step 2. Repeat at each generation $l$ until a stopping criterion is fulfilled
Step 3.     Select $N$ individuals ($D_l^{Se}$) from the $D_l$ population following a selection method
Step 4.     Induce from $D_l^{Se}$ an $n$ (size of the individual) dimensional probability model that
            shows the interdependencies between variables
Step 5.     Generate a new population $D_{l+1}$ of $M$ individuals based on
            the sampling of the probability distribution $p_l(x)$ learnt in the previous step
```

Fig. 1. Common outline for all Estimation of Distribution Algorithms (EDAs)

Steps 3, 4 and 5 will be repeated until a certain stop criterion is met (e.g., a maximum number of generations, a homogeneous population or no improvement after a specified number of generations).

The probabilistic model learnt at step 4 has a significant influence on the behavior of the EDA from the point of view of complexity and performance. EDAs are usually classified into three groups, attending to their ability to capture the dependencies between variables:

- Without dependencies: It is assumed that the n–dimensional joint probability distribution factorizes as a product of n univariate and independent probability distributions. Algorithms that use this model are, among others, Univariate Marginal Distribution Algorithm (UMDA) [40], Bit-Based Simulated Crossover (BSC) [70], compact Genetic Algorithm (cGA) [20], Population Based Incremental Learning in continuous domains ($PBIL_c$) [65] or Distribution Estimation Using MRF with direct sampling (DEUM) [67].
- Bivariate dependencies: Only the dependencies between pairs of variables are taken into account. This way, the process of estimating the joint probability can still be fast. This group includes Mutual Information Maximization for Input Clustering (MIMIC) [16], Bivariate Marginal Distribution Algorithm (BMDA) [56], or MIMIC approach to the continuous domain ($MIMIC_c$) [29,30].
- Multiple dependencies: All possible dependencies between the variables are considered, without taking into account the complexity of this process. In this group we can find algorithms like Estimation of Bayesian Networks Algorithm (EBNAs) [31], Estimation of Gaussian Networks Algorithms (EGNAs) [29, 30, 33], different flavors of the Bayesian Optimization Algorithm (BOA) [44, 50, 51, 52, 53], Learning Factorized Distribution Algorithm (LFDA) [41], Extended

Compact Genetic Algorithm (ECGA) [21], Markov Network Estimation of Distribution Algorithm (MN-EDA) [60] and Iterated Density Evolutionary Algorithms (IDEA) [6,7,8].

For detailed information about the characteristics of EDAs, and the algorithms that form part of this family, see [32,36,54,57].

2 EDAs and Parallelism

The advances that hardware platforms and software tools have experienced during the last decades have provided new, powerful tools to the research community, allowing researchers to tackle old problems using different approaches, or even to solve problems that were formerly unaffordable. In particular, the availability of powerful computational resources, such as clusters of computers, computational grids, graphics processing units and so on, has encouraged the development of parallel and distributed applications that can make use of these resources. Compared to their sequential counterparts, these parallel programs have shorter execution times, improve solution accuracy, or manage larger problems.

Before introducing the different parallelization proposals developed for EDAs, we provide a brief introduction to the most common computing platforms and programming paradigms available nowadays.

Most current server-class, desktop and laptop computers belong to the family of multiprocessors: a single board contains one or more processors, a certain amount of memory and other devices. Current processors contain two, four or more CPUs (cores). Multi-core processors are evolving towards many-core processors. Memory and other devices are shared among all the CPUs in the board. The potential of these machines can be fully exploited running several programs simultaneously (improving throughput) or running programs designed to carry out several tasks concurrently. The design of these programs require an API that allows the programmer to express how the tasks to perform are assigned to a collection of "threads", that communicate and synchronize between them. The most common APIs to these purposes are POSIX threads [11] and OpenMP [14]. Note that communication is implicit, via shared variables, and this mechanism is directly supported by the underlying hardware. Synchronization, required to access resources without conflicts, is explicit.

Clusters are another class of parallel computers. A cluster is composed of a set of independent computers (called nodes, which can be single- or multi-core) and an interconnection network that allows nodes to communicate and synchronize. The network can be a cheap Local Area Network (LAN), such as Ethernet, or a more expensive but better performing System Area Network (SAN), such as Infiniband or Myrinet. Note that nodes do not share memory; therefore, communication must be done via the network. Thread-based programming paradigms cannot be directly used in clusters. For these machines, the most common API is the Message Passing Interface [39]. MPI programs are expressed as independently-running processes that, when required, exchange messages between them. Synchronization can be explicit (barriers) or implicit (a receiving process blocks until a message arrives). When executing an MPI program, each process can run in a different cluster node.

MPI can be used in multiprocessors, using implementations that emulate message passing by means of shared variables. This is a convenient feature of MPI, because the same program can run in a multiprocessor, a cluster of single-processors, or a cluster of multiprocessors. However, when available, communication using shared variables is more efficient than message passing. For this reason programmers try to combine both paradigms, using threads for intra-node parallelism and MPI to communicate processes running in different nodes.

Computational grids extend the cluster concept, connecting machines (including clusters) using the Internet as the interconnection network. In theory, MPI applications can run, unmodified, in grid environments. However, there are a few considerations that we must bear in mind when deploying a parallel application on a grid: (1) the public Internet is not the best network in the world in terms of quality of service, which means that communication between processes may become a serious bottleneck, and (2) grid resources may not be available all the time, so applications are more prone to fail; programmers must introduce fault-tolerance mechanism to make applications more robust (and this fact can implicate a considerable overload).

In addition to a multi-core processor, many desktop computers include also a powerful Graphics Processing Unit (GPU). These sophisticated graphics rendering devices have a highly parallel structure, specifically designed for the efficient manipulation and display of graphics. But they can also be used to run other kinds of applications -sometimes at amazing speeds. GPU vendors provide APIs that allow programmers to harness the potential of their hardware, although they are not simple and require many tricks. Portability between different hardware platforms is another issue that has not been solved yet.

The modularity of EDAs makes this family of algorithms good candidates to be parallelized in any of the platforms discussed above. The parallelization approaches presented in the literature can be classified into these two groups:

- Direct parallelization: Those whose behavior is exactly the same of the corresponding sequential version. The main goal is the reduction of execution time, and the applicability to larger problems. Figure 2.a shows a (possible) general scheme, in which the program involves processes that play different roles. One of the processes is the Manager, in charge of driving the execution of the algorithm. The remaining processes play the Worker role, and devote their effort to compute the hardest parts of the algorithm.
- Island-based approach: Those that create different sub-populations and exchange information between them, trying to improve the behavior of the sequential algorithm (mainly in terms of the quality of the obtained solution). Figure 2.b shows the general scheme. In these algorithms, each island executes an "independent" EDA instance (processes may all run the same EDA, or different EDAs), exchanging at certain points information (individuals or dependency relations) with other islands, using a topology defined for that purpose (ring, star, etc.).

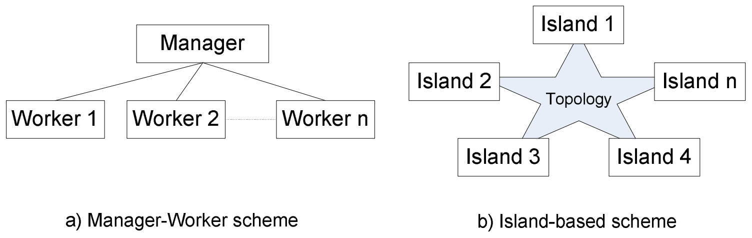


Fig. 2. Parallel schemes for EDAs

3 Direct Parallelization of Sequential EDAs

As stated previously, we call "direct parallelization" to the process of designing a parallel implementation of an original (sequential) EDA, with the goal of reducing the execution time, and without introducing any major change. This means that both the original and the parallel versions have the same behavior.

Several EDAs have been successfully applied to provide solutions for a range of complex problems. However, their application may become unfeasible if the individual size grows to a few hundred variables, or the fitness function is too complex. A fast and parallel EDA would provide two main advantages compared to the sequential counterpart: (1) it would allow the utilization of these approaches to solve larger or more complex problems in reasonable execution times, and (2) it would be able to complete a higher number of generations in the same execution time.

When parallelizing an existing sequential algorithm, it is mandatory to analyze carefully this sequential approach in order to detect if the algorithm is suitable to be parallelized. If this is the case, the parts that consume most of the execution time should be studied using profiling tools. Regarding EDAs, there are two steps that must be considered to design an efficient parallel algorithm: learning of the probabilistic model, and creation of the new population (sampling and evaluation of the individuals).

3.1 Learning of the Probabilistic Model

In Section 1 we introduced the general scheme of EDAs. Looking at the fourth step, we can observe that the main goal is to detect the possible (in)dependencies between the different variables. Probabilistic graphical models (PGMs) [26, 34, 49], which have become a popular representation for encoding uncertain knowledge, are commonly used to this purpose –they represent the joint probability distribution (the joint probability density function in the case of variables with continuous values).

PGMs use a graph to represent the (in)dependencies between the variables[1] (nodes) adding links between variables to represent such (in)dependencies.

[1] This term is equivalent to the term "gene" used in GAs.

Depending on the link, there are two types of PGMs. Those that use undirected links (edges) –for example Markov networks– and those that use directed links (arcs) –for example Bayesian networks [13] or Gaussian networks [66].

Learning a PGM requires two steps. Firstly, the graph that represents the relations between the variables is learnt; secondly, the probabilities (usually called parameters) are calculated. Depending on the method (complexity level) used to create PGMs from data, EDAs are classified into three groups: (1) all variables are considered independent, (2) only bivariate dependencies are taken into account, and (3) all dependencies are examined. In the following sections we present some techniques used in the previous groups, including helpful ideas from the point of view of parallelism[2].

3.1.1 Without Dependencies

The simplest way –first group– to estimate the joint probability distribution is to consider that all the variables are independent. Therefore, the joint probability distribution learnt at a given l generation from a set of selected individuals (D_l^{Se}) is:

$$p_l(x) = p(x|D_{l-1}^{Se}) = \prod_{i=1}^{n} p_l(x_i) \tag{1}$$

where each univariate marginal distribution is estimated from marginal frequencies:

$$p_l(x_i) = \frac{\sum_{j=1}^{N} \delta_j(X_i = x_i|D_{l-1}^{Se})}{N} \tag{2}$$

being

$$\delta_j(X_i = x_i|D_{l-1}^{Se}) = \begin{cases} 1 \text{ if in the } j^{th} \text{ case of } D_{l-1}^{Se}, X_i = x_i \\ 0 \text{ otherwise} \end{cases} \tag{3}$$

In this particular case, as all the variables are considered independent, the same fixed and linkless structure is used in every generation. Therefore, we only need to calculate the parameters. This is performed going over all the variables of the selected individuals.

In this case, the procedure of learning the PGM is simple and fast; therefore, it does not usually require any parallelization. However, if we wanted to write a parallel version of this procedure, there is a clear and easy way to distribute the work. The manager creates n subtasks, one for each variable to compute, and distributes these tasks (learn parameters) between the different workers (according to the scheme presented in Figure 2.a). In [61], the authors present an efficient and parallel version of a simple EDA (in particular, cGA) that is able to solve large-scale problems with millions of variables. To achieve this goal, the algorithm was parallelized following a manager-worker scheme with MPI, and was accurately designed (for example,

[2] By default we present proposals for the discrete domain, but these ideas and conclusions can be easily applied to the continuous domain.

using vectorization techniques so that a single instruction can perform multiple operations at the same time). This proposal clearly shows how helpful parallelism can be in this field.

Once the simplest probability model has been explained, it is time to go one step forward and focus on models that take into account (in)dependencies between pairs of variables, or even consider any kind of (in)dependencies. We will focus on the last group, because it is more general and includes the other one.

3.1.2 Multiple (in)dependencies

Different parallel proposals have been presented for the algorithms that belong to this group. These algorithms use as PGMs Markov networks, Gaussian networks, or Bayesian networks. In this case, we will focus on Bayesian networks to explain the main parallel approaches.

Formally, a Bayesian network [13] over a domain $\mathbf{X} = (X_1, \ldots, X_n)$ is a pair (S, θ) that represents a graphical factorization of a probability distribution. The structure S is a directed acyclic graph (DAG) which reflects the set of conditional (in)dependencies between the variables. The factorization of the probability distribution is written by:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{pa_i}), \tag{4}$$

where $\mathbf{pa_i}$ is the set of parents of X_i (variables from which there exists an arc to X_i in the graph S). The second part of the pair, θ, is a set of parameters for the local probability distributions associated with each variable. If variable X_i has r_i possible values, $x_i^1, \ldots, x_i^{r_i}$, the local distribution, $p(x_i|\mathbf{pa_i^j}, \theta_i)$ is an unrestricted discrete distribution:

$$p(x_i^k|\mathbf{pa_i^j}, \theta_i) \equiv \theta_{ijk}, \tag{5}$$

where $pa_i^1, \ldots, pa_i^{q_i}$ denote the values of $\mathbf{Pa_i}$ and the term q_i denotes the number of possible different instances of the parent variables of X_i. In other words, parameter θ_{ijk} represents the conditional probability of variable X_i being in its k^{th} value, knowing that the set of its parent variables is in its j^{th} value. Therefore, the local parameters are given by $\theta_i = (((\theta_{ijk})_{k=1}^{r_i})_{j=1}^{q_i})\ \ i = 1, \ldots, n$.

Two main techniques are used to learn the structure of the PGM –in this example, the Bayesian network. One is known as “score+search”, and the other as “detecting conditional (in)dependencies”.

“score+search”

This method is based on two components: a score metric and a search algorithm. The score metric is used to measure the quality of the Bayesian network structure, given a data file of cases. Bayesian Information Criterion (BIC) [63], K2 [15] or Bayesian Dirichlet equivalence (BDe) [22] are some of the scores commonly used by the community. The search algorithm is required to look for the structure that maximizes the selected score. For example, Algorithm B [10] is a common method used to learn Bayesian networks. This algorithm uses a hill climbing strategy. Starting with an arc-less structure, it adds in each step the arc that maximizes the score. When no improvement can be achieved, the algorithm stops. An alternative to Algorithm B could be the use of the model created in the previous generation, instead of beginning each time with an empty structure. Other proposals try to improve the search by including the arc reversal operator, in addition to add and delete.

An important property of the BIC, K2, and DBe scores (from the point of view of parallelism) is that they are decomposable. This means that the global score can be calculated as the sum of the separate local scores for the variables. Focusing on the BIC score, it can be expressed as:

$$BIC(S,D) = \sum_{i=1}^{n} \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log \frac{N_{ijk}}{N_{ij}} - \frac{1}{2} \log N \sum_{i=1}^{n} q_i (r_i - 1), \tag{6}$$

where:

- S is the structure and D is a dataset (set of selected individuals).
- n is the number of variables of the Bayesian network (size of the individual).
- r_i is the number of different values that variable X_i can take.
- q_i is the number of different values that the parent variables of X_i, Pa_i, can take.
- N_{ij} is the number of individuals in D in which variables Pa_i take their j^{th} value.
- N_{ijk} is the number of individuals in D in which variable X_i takes its k^{th} value and variables Pa_i take their j^{th} value.

According to its decomposable property, it can be expressed as the sum of the local scores for each variable X_i:

$$BIC(S,D) = \sum_{i=1}^{n} BIC(i,S,D), \tag{7}$$

where

$$BIC(i,S,D) = \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log \frac{N_{ijk}}{N_{ij}} - \frac{1}{2} \log(N) q_i (r_i - 1). \tag{8}$$

Due to this, if we update S with the arc modification (j,i), then only $BIC(i,S,D)$ needs to be recalculated.

The structural learning algorithm involves a sequence of actions that differs between the first step and all the subsequent steps. In the first step, given a structure S and a database D, the change in the BIC is calculated for each possible arc modification. Thus, we have to calculate $n(n-1)$ terms as there are $n(n-1)$ possible arc modifications. The arc modification that maximizes the gain of the BIC score, whilst maintaining the DAG structure, is applied to S. Supposing that an arc from node j to node i was added, only the local BIC score for node i has to be calculated. Other local scores have not changed its value because of the decomposable property of the BIC score. In this case, the number of terms to be calculated is $n-2$. Figs. 3 and 4 show a parallel approach (introduced in [35, 38]) for this learning process.

```
Manager side. "score+search" technique.

Step 1. for $i = 1, \ldots, n$
            Send order Calculate $BIC(i,S,D)$
            Receive local scores from workers
Step 2. if score can be improved
            Modify arc $(j,i)$
            Send order Modify arc $(j,i)$
            Send order Calculate $BIC(i,S,D)$
            Go to Step 2
        else Send order Stop
Step 3. End
```

Fig. 3. Pseudo-code for the parallel structural learning based on "score+search". Manager side

```
Worker side. "score+search" technique.

Step 1. Define the set of variables to work with ($N_{wrk}$)
Step 2. wait for an order
Step 3. case order of
            Calculate $BIC(i,S,D)$
                for each j in $N_wrk$
                    Calculate new BIC score for the $(j,i)$ arc modification
                Send new scores to the manager
                Go to Step 2
            Modify arc $(j,i)$
                Modify arc $(j,i)$
                Go to Step 2
            Stop
                End
```

Fig. 4. Pseudo-code for the parallel structural learning based on "score+search". Worker side

Regarding this proposal, there are some aspects that deserve additional detail:

- In order to avoid unbalance, in homogenous environments each worker should be assigned an equal-size subset of the nodes. However, in heterogeneous environments (a variety of hardware elements with different processing power) an on-demand scheme would be more appropriate: each worker would ask repeatedly for a task to complete, until there are no more tasks. Obviously, the on-demand scheme would make a more intensive use of the communication infrastructure.
- In order to achieve better performances, the manager process could also act as a worker. Looking at the pseudo-code, we can observe how the manager sends

the order to calculate BIC scores and then waits idly until all the workers have finished. Instead of just waiting, the manager could become an additional worker.
- As all the workers need to know the structure, some communication between manager and workers is needed. As a general rule, in order to reach good levels of performance from distributed-memory parallel computers, communication should be minimized -specially if the network is slow. However, experiments carried out on homogenous clusters showed that communication time is tiny, compared to the time needed to compute the BIC scores.

Table 1, extracted from [38], shows results for the *OneMax* problem [62], with an individual size of 500. The machine used to carry out the experiments was a cluster of 10 nodes. Each node had two AMD ATHLON MP 2000+ processors (CPU frequency 1.6GHz) and 1GB of RAM. The operating system was GNU-Linux and the MPI implementation was LAM (6.5.9. version). All the computers were interconnected using a switched Gigabit Ethernet network.

Good levels of efficiency are observed, when using up to 20 processors. In this approach it is important to find a balance between the size (complexity) of the problem and the number of processors used. That is, as the number of processors increases, the work (number of variables) that each worker manages decreases, and hence, the relation between computation and communication worsens.

Table 1. Time-related experimental results for two versions of the $EBNA_{BIC}$ parallel algorithm

CPUs	*MPI version*			*MPI&Threads version*		
	Time	*Speed Up*	*Efficiency*	*Time*	*Speed Up*	*Efficiency*
Sequential	2h 25' 42"	-	-	-	-	-
2	1h 08' 44"	2.12	1.06	1h 08' 13"	2.14	1.07
6	24' 38"	5.91	0.99	24' 23"	5.97	1.00
10	15' 21"	9.49	0.95	15' 32"	9.38	0.94
20				08' 54"	16.39	0.82

In summary, this is a good approach to compute the score in a distributed fashion, but there must be taken into account the recommendations recently discussed about communication requirements in distributed-memory machines where the programming paradigm is, usually, MPI. This problem disappears for implementations based on threads or OpenMP, but these paradigms are limited to shared-memory computers.

Related to the need of minimizing communications when learning the PGM, [45] discusses a different approach for the MBOA algorithm (based on decision trees) that avoids communication while creating the structure. Each worker learns a part of the structure in an asynchronous (independent) way. To maintain the structure acyclic without any communication, the authors propose performing, at each generation, a random permutation that predetermines the topological order of the nodes. They claim, using experimental evidence [55], that this a-priori order does not affect the overall behavior of the algorithm.

"detecting conditional (in)dependencies"

Techniques in this group use independence tests to check the relations between the variables in the structure. They usually start from a complete graph, verifying whether or not a link should be maintained. A good review of methods for the induction of Bayesian networks by detecting conditional (in)dependencies can be found in [69]. From the point of view of parallelism, a usual drawback of these tests is that they can not be easily distributed in separated subsets (one for each worker); frequent synchronization is needed, because the result of one test conditions the following tests. Fig. 5, extracted from [38], shows some results obtained using the *PC* algorithm to detect conditional (in)dependencies in the same problem introduced for the "score+search" method (*OneMax*, with an individual size of 500). Implementation was done using MPI, following an on-demand scheme. It can be observed that communication becomes an important bottleneck as the number of processors increases. Again, if the (parallel) computer allows it, an implementation with threads or OpenMP would reduce the communication effort. In general, for very communication-demanding algorithms, it is a good idea to mix MPI with threads or OpenMP to make a more efficient use of today's multi-core computers.

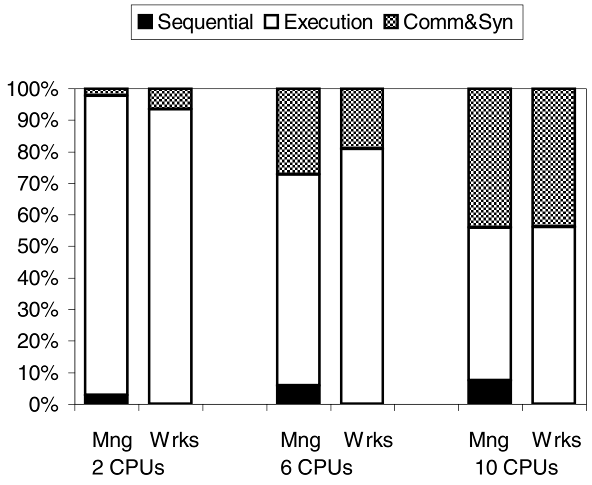


Fig. 5. Detail of the computation time for the second version of the $EBNA_{PC}$ algorithm, using a pure MPI implementation. 2, 6 and 10 CPUs have been used.

3.2 Sampling and Evaluation of the Population

The fifth step of EDAs, that involves sampling the probability distribution to generate new individuals, and evaluating the new population, is computationally expensive when dealing with non-trivial evaluation functions. Therefore, this step must be parallelized. In [12] the author introduces a summary of ideas for the design of efficient parallel evolutionary algorithms, that includes parallelization of fitness evaluation procedures. For EDAs the same principles apply. Additionally, the sampling part (creation of the individuals) can also be done in a distributed way.

Once the PGM has been learnt (structure and parameters) –in our example a Bayesian network–, new individuals must be generated by means of the joint probability distribution encoded by the network. To do that, individuals will be generated by sampling directly from the Bayesian network, using an adaptation of the Probabilistic Logic Sampling algorithm (PLS) [23]. Once generated, individuals must be evaluated using a problem-dependent fitness function. This sampling process is repeated, creating and evaluating in each iteration a new individual, until the required number of new individuals is reached.

From the point of view of parallelism, a very simple manager-worker approach can be implemented: the manager sends the parameters of the Bayesian network to the workers (assuming that they know the structure), they sample (create) a subset of new individuals, evaluate them, and send back the subset to the manager. Again, a homogeneous computing environment is assumed, so all subsets are of equal size, but an on-demand scheme (in which workers ask the manager for tasks to perform, and return new, evaluated individuals) could be more appropriate for heterogeneous systems.

4 Island-Based Approaches

Although ideas suggesting the utilization of parallel approaches for EAs were discussed in different papers long time ago [9, 25], it has been only in the last two decades when the availability of hardware and software has made possible the design of usable parallel solutions. Sequential EAs deal with a single population, in which each of the individuals can potentially mate with any other. Using this starting point, two parallel proposals appear: distributed EAs (dEAs) and cellular EAs (cEAs).

dEAs are known as coarse grain parallel EAs. Algorithms of this kind use several sub-populations (islands), instead of only one. These populations evolve in a quasi-independent way and, with some predetermined frequency (usually a number of generations), some individuals are exchanged between islands. Different topologies have been tested for different problems, including for example rings and stars. When implemented in a parallel computer, each island can be mapped onto a processor.

Regarding cEAs (also known as fine grain parallel EAs), individuals are generally placed on a toroidal n-dimensional grid (one at each position). Every individual has a neighborhood, and the breeding operators will be applied between the individuals in the neighborhood. When implemented in a parallel computer, each individual of the current population has to be mapped onto a processor. cEAs are particularly well-suited for massively parallel computing systems, while dEAs can run efficiently in cluster-type systems.

Figure 6 shows the distribution of the populations in sequential EAs, dEAs and cEAs. These ideas have been widely applied to GAs; a summary of different parallel techniques for GAs can be found in [12]. In addition, a review about parallelism and EAs, and an up-to-date reference on parallel metaheuristics can be consulted in [4] and [2] respectively.

dEAs are of particular interest due to the utilization of several sub-populations – called islands– that evolve independently, exchanging some individuals from time to time. In addition to the parameters commonly used in GAs (such as population size or breeding operators), these island-based approaches need to fix new parameters:

- Number of islands: Controls the number of sub-populations that will be used. This number is usually related to the number of available processors.
- Number of individuals exchanged at each migration: Determines the number of individuals that an island will receive from another island (or islands). The larger this number is, the more similar the sub-populations will be.
- Migration rate: Indicates how often migrations are made to happen. Usually, this parameter if expressed in terms of the number of generations between migrations. However, other proposals can be found in the literature that are based on asynchronous communication [5].
- Topology: Defines the connections between islands: ring, star, grid, and so on.
- Replacement: When a subset of individuals is received in an island, there are different ways to incorporate them to the present population. The most common technique is to apply elitism, that is, the subset is added to the population, and then the worst individuals of the mixed population are deleted.

The main purpose of island-based proposals is to preserve the diversity of the whole population splitting it in different sub-populations, trying to avoid a premature convergence. When a good solution appears in an island, some generations will

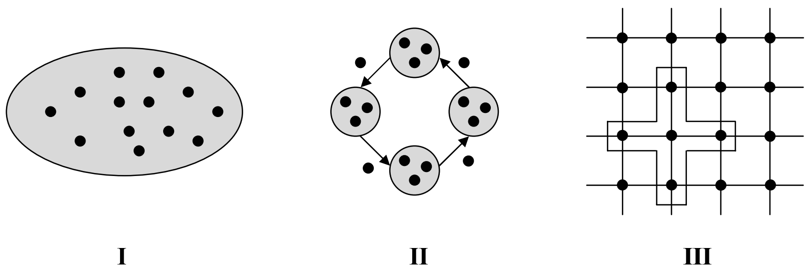


Fig. 6. (I) sequential EA, in which all individuals belong to the same population, (II) dEA, in which individuals are distributed in a number of sub-populations, and (III) cEA with individuals distributed in a grid.

pass until that solution arrives to other islands, allowing this way the search in other areas of the solution space. However, even if there are several works on this issue [3,12,71] there is not still a clear idea of how these approaches perform in terms of quality of the results or convergence rate.

Following the ideas developed for GAs, island-based approaches have also been used with EDAs, for discrete [1, 37, 43, 59] as well as for continuous domains [24, 37]. In [24], the authors propose the Distributed Probabilistic Model Building GA (DPMBGA), a new EDA approach that uses Principal Component Analysis to detect dependencies between the variables and uses normal distributions to create new individuals. The model exchanges individuals over a ring topology. This algorithm has been extended in [68], introducing the penalty method and the pulling back method in order to improve the characteristics of the previous version when solving constraint problems. In [37], an asynchronous distributed framework for EDAs is presented and implemented for the UMDA algorithm.

The particular characteristics of EDAs allow researchers to explore additional research lines. One idea is to use a different EDA in each island [2]. In this way, the weakness of one algorithm could be compensated by the advantages of others. However, running different algorithms implies different execution times per iteration (generation) and, therefore, the migration method should be adapted (usually, by means of asynchronous protocols).

Another idea lays down the most representative characteristic of EDAs: the use of probabilistic models. Taking into account that the probabilistic model comprises the relevant information about the problem, it should be better to exchange that knowledge instead of exchanging a few individuals. In [1], the authors use a univariate EDA, exchanging the probability vector between the islands. In each island, the probability vectors are combined by means of a crossover operator. In [46] the authors design different island-based (dEAs) topologies, and carry out some experiments using two different types of migration: individuals and probabilistic models. For the latter approach (exchange of probabilistic models), a univariate EDA for the discrete domain is used and, in the migration step, each island mixes its current probabilistic model with the received one. In that preliminary work, the researchers propose a convex combination of the models (similar to that proposed in PBIL), while in a later paper [47], the model combination is improved using a local search. Results obtained from experiments show that, in general, approaches that export and combine probabilistic models provide better solutions than those that exchange individuals. Furthermore, solutions provided by the best island-based EDAs are often superior to those obtained with the sequential, single-population counterpart. This advantage comes from the availability of different, partially isolated populations, allowing the algorithm to search in different parts of the solution space. Additionally, diversity is higher, avoiding premature convergence. These conclusions have been verified in other similar works based on the migration and combination of the probability models [27, 64]. The same ideas have also been adapted to the continuous domain in [48], translating the proposals to the $UMDA_c$ algorithm.

In summary, this is a new field of research inside the EDA family, and there are still some aspects that need further work:

- Island-based approaches need additional parameters. It is not clear which the best combination of values in order to properly tune an island-based model is. For example, it is necessary to decide the topology to be used, the number of islands, the migration scheme, and so on. According to the results, some general ideas can be obtained but, generally, the values of the parameters seem to be problem dependent. In addition, there are some voices that impeach the alleged goodness of these models.
- Approaches that exchange probability models, instead of groups of individuals, seem more promising. Probability models comprise more information than a few individuals. This information is easier to analyze, and these models are also able to provide information about the (in)dependencies between the variables.
- The proposals discussed in this section have been tested using algorithms with simple (univariate) probabilistic models. These models are able to obtain good results in a wide range of scenarios but, for problems where there are dependencies between the variables, they usually behave worse than more complex probabilistic models. The extension of methods that combine probabilistic models to algorithms that use pairwise or multiple-dependency models is still an open line of research.
- In terms of computing performance, island models can help to reduce execution times, because the islands run in parallel, each one with a subset of the population. The size of the population plays a crucial role in EDAs based on complex models (multiple dependencies): the larger the population, the better the obtained solutions. Therefore, splitting the whole population in several sub-populations will have, in theory, a negative impact on the quality of the solutions. Several, large populations could be used, but then the parallel EDA would not run faster than a single-population, sequential one.
- Some works report that island-based approaches can achieve faster convergence rates, thus reducing the number of generations (evaluations) required to find the desired solutions.

As a final remark, we want to state that the two main approaches discussed through this chapter (direct parallelism and island-based approaches) should not be seen as incompatible alternatives. In fact, they can be combined, using islands that run accelerated EDAs. Most current parallel computers are actually clusters of multiprocessors, and are well suited to run this kind of solutions. For example, the program implementing an island could be composed of several threads running in different cores of a cluster node; islands would communicate through an external, high-speed interconnection network using MPI calls.

5 Conclusions

In this chapter we have made a review of the different proposals that have appeared in the literature to parallelize Estimation of Distribution Algorithms (EDAs). These proposals can be classified into two groups: direct parallelism and island-based approaches.

Direct parallelism entails the design of parallel versions of sequential EDAs with the aim of reducing the execution time, while maintaining the behavior of the original algorithm. Design effort is focused on two phases of the EDAs: the learning of the probabilistic model, and the sampling and evaluation of the population –specially, to the evaluation part. These implementations have been shown to effectively reduce the total execution time, allowing the application of EDAs to large-scale problems.

Island-based approaches use several sub-populations (islands), instead of the usual single population. Sub-populations evolve independently, using a given EDA (that is usually the same for all islands, but could also be different). At certain moments, groups of individuals or probability models are exchanged between the islands. This exchange is done following a particular communication topology: rings, stars, point to point, and so on. Preliminary work on island-based distributed EDAs show promising results in terms of execution speed and quality of solutions, but many research lines are still open in this field.

As a summary, this is a line of work that has helped EDAs to go a step forward in terms of usability. The increasingly easier (and cheaper) access to multi-core (even many-core) processors, clusters of computers, grid environments, etc. are encouraging researchers to make use of parallel paradigms in order to develop faster and more efficient programs that solve larger and more complex problems.

Acknowledgements. This work has been partially supported by the Etortek, Saiotek and Research Groups 2007-2012 (IT-242-07) programs (Basque Government), TIN2007-68023-C02-02, TIN2005-03824 and Consolider Ingenio 2010 - CSD2007-00018 projects (Spanish Ministry of Education and Science) and COMBIOMED network in computational biomedicine (Carlos III Health Institute).

References

1. Ahn, C.W., Goldberg, D.E., Ramakrishna, R.: Multiple-deme parallel estimation of distribution algorithms: Basic framework and application. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2004. LNCS, vol. 3019, pp. 544–551. Springer, Heidelberg (2004)
2. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. John Wiley & Sons Inc., Chichester (2005)
3. Alba, E., Cotta, C., Troya, J.: Numerical and real-time analysis of parallel distributed GAs with structured and panmictic populations. In: Proceedings of the IEEE Conference on Evolutionary Computing (CEC), vol. 2, pp. 1019–1026 (1999)
4. Alba, E., Tomassini, M.: Paralelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation 6(5), 443–462 (2002)
5. Alba, E., Troya, J.M.: An analysis of synchronous and asynchronous parallel distributed genetic algorithms with structured and panmictic islands. In: Rolim, J.D.P., Mueller, F., Zomaya, A.Y., Erçal, F., Olariu, S., Ravindran, B., Gustafsson, J., Takada, H., Olsson, R.A., Kalé, L.V., Beckman, P.H., Haines, M., ElGindy, H.A., Caromel, D., Chaumette, S., Fox, G., Pan, Y., Li, K., Yang, T., Ghiola, G., Conte, G., Mancini, L.V., Méry, D., Sanders, B.A., Bhatt, D., Prasanna, V.K. (eds.) IPPS-WS 1999 and SPDP-WS 1999. LNCS, vol. 1586, pp. 248–256. Springer, Heidelberg (1999)

6. Bosman, P.A.N.: Design and application of iterated density-estimation evolutionary algorithms. Ph.D. thesis, Utrech University (2003)
7. Bosman, P.A.N., Thierens, D.: An algorithmic framework for density estimation based evolutionary algorithms. Tech. Rep. UU-CS-1999-46, Utrech University (1999)
8. Bosman, P.A.N., Thierens, D.: IDEAs bases on the normal kernels probability density function. Tech. Rep. UU-CS-2000-11, Utrech University (2000)
9. Bossert, W.: Mathematical optimization: Are there abstract limits on natural selection? In: Moorehead, P.S., Kaplan, M.M. (eds.) Mathematical Challenges to the Neo-Darwinian Interpretation of Evolution, pp. 35–46. The Wistar Institute Press, Philadelphia (1967)
10. Buntine, W.: Theory refinement in Bayesian networks. In: Proceedings of the Seventh Conference on Uncertainty in Artificial Intelligence, pp. 52–60 (1991)
11. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Professional Computing Series (1997)
12. Cantú-Paz, E.: Efficient and accurate parallel genetic algorithms. Kluwer Academic Publishers, Dordrecht (2000)
13. Castillo, E., Gutiérrez, J.M., Hadi, A.S.: Expert Systems and Probabilistic Network Models. Springer, New York (1997)
14. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). MIT Press, Cambridge (2007)
15. Cooper, G.F., Herskovits, E.A.: A Bayesian method for the induction of probabilistic networks from data. Machine Learning 9, 309–347 (1992)
16. De Bonet, J.S., Isbell, C.L., Viola, P.: MIMIC: Finding optima by estimating probability densities. In: Mozer, M., Jordan, M., Petsche, T. (eds.) Advances in Neural Information Processing Systems, vol. 9 (1997)
17. Fogel, L.J.: Autonomous automata. Ind. Res. 4, 14–19 (1962)
18. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison/Wesley, Reading (1989)
19. Grefenstette, J.J.: Optimization of control parameters for genetic algorithms. IEEE Transactions on Systems, Man, and Cybernetics 16(1), 122–128 (1986)
20. Harik, G.R., Lobo, F.G., Goldberg, D.E.: The compact Genetic Algorithm. IEEE Transactions on Evolutionary Computation 3(4), 287–297 (1999)
21. Harik, G.R., Lobo, F.G., Sastry, K.: Linkage learning via probabilistic modeling in the extended compact genetic algorithm (ecga). In: Pelikan, M., Sastry, K., Cantú-Paz, E. (eds.) Scalable Optimization via Probabilistic Modeling. Studies in Computational Intelligence, vol. 33, pp. 39–61. Springer, Heidelberg (2006)
22. Heckerman, D., Geiger, D., Chickering, D.M.: Learning Bayesian networks: The combination of knowledge and statistical data. Machine Learning 20, 197–243 (1995)
23. Henrion, M.: Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In: Lemmer, J.F., Kanal, L.N. (eds.) Uncertainty in Artificial Intelligence, vol. 2, pp. 149–163. North-Holland, Amsterdam (1988)
24. Hiroyasu, T., Miki, M., Sano, M., Shimosaka, H., Tsutsui, S., Dongarra, J.: Distributed Probabilistic Model-Building Genetic Algorithm. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L., Roy, R., O'Reilly, U.M., Beyer, H.G., Standish, R.K., Kendall, G., Wilson, S.W., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A.C., Dowsland, K.A., Jonoska, N., Miller, J.F. (eds.) GECCO 2003. LNCS, vol. 2723, pp. 1015–1028. Springer, Heidelberg (2003)
25. Holland, J.H.: Adaptation in Natural and Artificial Systems. The University of Michigan Press (1975)

26. Howard, R., Matheson, J.: Influence diagrams. In: Howard, R., Matheson, J. (eds.) Readings on the Principales and Applications of Decision Analysis, vol. 2, pp. 721–764. Strategic Decision Group, Menlo Park (1981)
27. Jaros, J., Schwarz, J.: Parallel BMDA with probability model migration. In: Proceeding of 2007 IEEE Congress on Evolutionary Computation, pp. 1059–1066. IEEE Computer Society, Los Alamitos (2007)
28. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
29. Larrañaga, P., Etxeberria, R., Lozano, J., Peña, J.: Optimization by learning and simulation of Bayesian and Gaussian networks. Tech. Rep. KZZA-IK-4-99, Department of Computer Science and Artificial Intelligence, University of the Basque Country (1999)
30. Larrañaga, P., Etxeberria, R., Lozano, J., Peña, J.: Optimization in continuous domains by learning and simulation of Gaussian networks. In: Proceedings of the Workshop in Optimization by Building and using Probabilistic Models. A Workshop within the 2000 Genetic and Evolutionary Computation Conference, GECCO 2000, Las Vegas, Nevada, USA, pp. 201–204 (2000)
31. Larrañaga, P., Etxeberria, R., Lozano, J.A., Peña, J.M.: Combinatorial optimization by learning and simulation of Bayesian networks. In: Proceedings of the Conference on Uncertainty in Artificial Intelligence, UAI 2000, Stanford, CA, USA, pp. 343–352 (2000)
32. Larrañaga, P., Lozano, J.A.: Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation. Kluwer Academic Publishers, Dordrecht (2002)
33. Larrañaga, P., Lozano, J.A., Bengoetxea, E.: Estimation of Distribution Algorithms based on multivariate normal and Gaussian networks. Tech. Rep. KZZA-IK-1-01, Department of Computer Science and Artificial Intelligence, University of the Basque Country (2001)
34. Lauritzen, S.L.: Graphical Models. Oxford University Press, Oxford (1996)
35. Lozano, J., Sagarna, R., Larrañaga, P.: Parallel estimation of distribution algorithms. In: Larrañaga, P., Lozano, J.A. (eds.) Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation, pp. 129–145. Kluwer Academic Publishers, Dordrecht (2002)
36. Lozano, J.A., Larrañaga, P., Inza, I., Bengoetxea, E.: Towards a New Evolutionary Computation. In: Advances on Estimation of Distribution Algorithms. Studies in Fuzziness and Soft Computing. Springer, New York (2006)
37. Madera, J., Alba, E., Ochoa, A.: A Parallel Island Model for Estimation of Distribution Algorithms. In: Lozano, J.A., Larrañaga, P., Inza, I., Bengoetxea, E. (eds.) Towards a New Evolutionary Computation. Advances on Estimation of Distribution Algorithms. Studies in Fuzziness and Soft Computing, vol. 192, pp. 159–186. Springer, Heidelberg (2005)
38. Mendiburu, A., Lozano, J.A., Miguel-Alonso, J.: Parallel implementation of EDAs based on probabilistic graphical models. IEEE Transactions on Evolutionary Computation 9(4), 406–423 (2005)
39. Message Passing Interface Forum: MPI: A message-passing interface standard. International Journal of Supercomputer Applications (1994)
40. Mühlenbein, H.: The equation for response to selection and its use for prediction. Evolutionary Computation 5, 303–346 (1998)
41. Mühlenbein, H., Mahning, T.: FDA - a scalable evolutionary algorithm for the optimization of additively decomposed functions. Evolutionary Computation 7(4), 353–376 (1999)

42. Mühlenbein, H., Paaß, G.: From recombination of genes to the estimation of distributions i. binary parameters. In: Ebeling, W., Rechenberg, I., Voigt, H.-M., Schwefel, H.-P. (eds.) PPSN 1996. LNCS, vol. 1141, pp. 178–187. Springer, Heidelberg (1996)
43. Ocenasek, J.: Parallel estimation of distribution algorithms. Ph. D. thesis, Faculty of Information Technology, Brno University of Technology (2002)
44. Ocenasek, J., Schwarz, J.: Estimation of distribution algorithm for mixed continuous-discrete optimization problems. In: 2nd Euro-International Symposium on Computational Intelligence, pp. 227–232. IOS Press, Kosice (2002)
45. Ocenasek, J., Schwarz, J., Pelikan, M.: Design of multithreaded estimation of distribution algorithms. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Kendall, G., Wilson, S.W., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A., Dowsland, K.A., Jonoska, N., Miller, J., Standish, R.K. (eds.) GECCO 2003. LNCS, vol. 2724, pp. 1247–1258. Springer, Heidelberg (2003)
46. De la Ossa, L., Gámez, J.A., Puerta, J.M.: Migration of probability models instead of individuals: An alternative when applying the island model to EDAs. In: Yao, X., Burke, E.K., Lozano, J.A., Smith, J., Merelo-Guervós, J.J., Bullinaria, J.A., Rowe, J.E., Tiňo, P., Kabán, A., Schwefel, H.-P. (eds.) PPSN 2004. LNCS, vol. 3242, pp. 242–252. Springer, Heidelberg (2004)
47. De la Ossa, L., Gámez, J.A., Puerta, J.M.: Improving model combination through local search in parallel univariate edas. In: Congress on Evolutionary Computation, pp. 1426–1433. IEEE, Los Alamitos (2005)
48. De la Ossa, L., Gámez, J.A., Puerta, J.M.: Initial approaches to the application of islands-based parallel EDAs in continuous domains. In: Skie, T., Yang, C.S. (eds.) ICPP Workshops, pp. 580–587. IEEE Computer Society, Los Alamitos (2005)
49. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, Palo Alto (1988)
50. Pelikan, M., Goldberg, D.E.: Genetic algorithms, clustering, and the breaking of symmetry. In: Deb, K., Rudolph, G., Lutton, E., Merelo, J.J., Schoenauer, M., Schwefel, H.-P., Yao, X. (eds.) PPSN 2000. LNCS, vol. 1917. Springer, Heidelberg (2000)
51. Pelikan, M., Goldberg, D.E.: Hierarchical problem solving and the Bayesian optimization algorithm. In: Whitley, D., Goldberg, D., Cantú-Paz, E., Spector, L., Parmee, I., Beyer, H.G. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, vol. 1, pp. 267–274. Morgan Kaufmann Publishers, San Francisco (2000)
52. Pelikan, M., Goldberg, D.E.: Research on the Bayesian optimization algorithm. In: Wu, A. (ed.) Proceedings of the 2000 Genetic and Evolutionary Computation Conference Workshop Program, vol. 1, pp. 212–215 (2000)
53. Pelikan, M., Goldberg, D.E., Cantú-Paz, E.: BOA: The Bayesian optimization algorithm. In: Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference GECCO 1999, Orlando FL, vol. 1, pp. 525–532. Morgan Kaufmann Publishers, San Francisco (1999)
54. Pelikan, M., Goldberg, D.E., Lobo, F.: A survey of optimization by building and using probabilistic models. Computational Optimization and Applications 21(1), 5–20 (2002)
55. Pelikan, M., Laury Jr., J.D.: Order or not: does parallelization of model building in hboa affect its scalability? In: Lipson, H. (ed.) Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, pp. 555–561. ACM, New York (2007)
56. Pelikan, M., Mühlenbein, H.: The bivariate marginal distribution algorithm. In: Roy, P.K.C.R., Furuhashi, T. (eds.) Advances in Soft Computing-Engineering Design and Manufacturing, pp. 521–535. Springer, London (1999)

57. Pelikan, M., Sastry, K., Cantú-Paz, E.: Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications. Studies in Computational Intelligence. Springer, New York (2006)
58. Rechenberg, I.: Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Frommann–Holzboog, Stuttgart (1973)
59. Robles, V., Perez, M., Peña, J., Herves, V., Larrañaga, P.: Parallel interval estimation naive bayes. In: Actas de las XIV Jornadas de Paralelismo, pp. 349–353 (2003)
60. Santana, R.: Estimation of distribution algorithms with Kikuchi approximations. Evolutionary Computation 13(1), 67–97 (2005)
61. Sastry, K., Goldberg, D.E., Llorà, X.: Towards billion-bit optimization via a parallel estimation of distribution algorithm. In: Lipson, H. (ed.) Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, pp. 577–584. ACM, New York (2007)
62. Schaffer, J.D., Eshelman, L.J.: On crossover as an evolutionarily viable strategy. In: Belew, R.K., Booker, L.B. (eds.) ICGA, pp. 61–68. Morgan Kaufmann, San Francisco (1991)
63. Schwarz, G.: Estimating the dimension of a model. Annals of Statistics 7(2), 461–464 (1978)
64. Schwarz, J., Jaros, J., Ocenásek, J.: Migration of probabilistic models for island-based bivariate eda algorithm. In: 2007 Genetic and Evolutionary Computational Conference, vol. I, p. 631. Association for Computing Machinery (2007)
65. Sebag, M., Ducoulombier, A.: Extending Population-Based Incremental Learning to Continuous Search Spaces. In: Eiben, A.E., Bäck, T., Schoenauer, M., Schwefel, H.-P. (eds.) PPSN 1998. LNCS, vol. 1498, pp. 418–427. Springer, Heidelberg (1998)
66. Shachter, R., Kenley, C.: Gaussian influence diagrams. Management Science 35, 527–550 (1989)
67. Shakya, S., McCall, J., Brown, D.: Updating the probability vector using MRF technique for a univariate EDA. In: Onaindia, E., Staab, S. (eds.) STAIRS 2004, Proceedings of the Second Starting AI Researchers' Symposium. Frontiers in Artificial Intelligence and Applications, vol. 109. IOS Press, Valencia (2004)
68. Shimosaka, H., Hiroyasu, T., Miki, M.: Comparison of pulling back and penalty methods for constraints in BGA. In: Sarker, R., Reynolds, R., Abbass, H., Tan, K.C., McKay, B., Essam, D., Gedeon, T. (eds.) Proceedings of the 2003 Congress on Evolutionary Computation, CEC 2003, pp. 1941–1948. IEEE Press, Canberra (2003)
69. Spirtes, P., Glymour, C., Scheines, R.: Causation, Prediction, and Search. Lecture Notes in Statistics, vol. 81. Springer, Heidelberg (1993)
70. Syswerda, G.: Simulated crossover in genetic algorithms. In: Foundations of Genetic Algorithms, vol. 2, pp. 239–255. Morgan Kaufmann, San Mateo (1993)
71. Whitley, L.D., Rana, S.B., Heckendorn, R.B.: Island model genetic algorithms and linearly separable problems. In: Corne, D., Shapiro, J.L. (eds.) Evolutionary Computing, AISB Workshop. LNCS, vol. 1305, pp. 109–125. Springer, Heidelberg (1997)
72. Zhigljavsky, A.A.: Theory of Global Random Search. Kluwer Academic Publishers, Dordrecht (1991)

Parallel Multi-objective Optimization Using Self-organized Heterogeneous Resources

Sanaz Mostaghim

1 Summary

This chapter is about using a set of parallel self-organized computing resources to perform multi-objective optimization. These computing resources are presented as a unified resource to the user where in the traditional parallel optimization paradigms the user has to assign tasks to the resources, collect the best available solutions and deal with failing resources. The main goal in this chapter is to involve the user as less as possible in the optimization process. Here the user only specifies the preferences and gives the objective functions to the system. The self-organized computing resources deliver the obtained solutions after a certain time to the user. In such a system, fast resources continue the optimization as long as the overall computing time is not over. However as the solutions of a multi-objective problem depend on each other (via the domination relation) adding a waiting time to the fast processors would affect the quality of the solutions. This has been studied on a scenario of 100 heterogeneous computing resources in the presence of failures in the system.

2 Introduction

Due to the steady progress in technology and the fact that the number of computing resources is increasing, today parallel computing on computer grids or multi-core systems can significantly reduce the computation time for highly complex modeling, simulation, and optimization problems.

Sanaz Mostaghim
Institute AIFB
University of Karlsruhe
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
e-mail: sanaz.mostaghim@kit.edu

Generally, distributed and parallel computing are about using several computing resources in parallel in order to solve time-intensive problems in a low computation time. Ideally, these computing resources are represented as one unified resource to the user who defines the task to be run in parallel [1]. A typical task can be a simulation with different input parameters or an optimization algorithm which can be run in parallel. In such a computing environment we deal with a set of heterogeneous resources. There are very fast and very slow computing resources, some computing resources may fail working, or get overloaded. Also, we assume that there is a limitation on the communication capability between the resources.

In the context of optimization, there are several reasons for using parallel computing, such as:

- in order to solve optimization problems with very time intensive function evaluations,
- in order to solve large scale problems,
- some optimization algorithms require a high computation time in order to deliver some reasonable solutions,
- when there are several available computing resources and the user wants to use all of them even for simple optimization tasks.

In optimization, different parallel models have been proposed in the literature [23]. They follow three major hierarchical models such as: (1) Self-contained parallel cooperation (2) Problem independent intra-algorithm parallelization, and (3) Problem dependent intra-algorithm parallelization. The last two models do not alter the behavior of the algorithms and therefore are generally used to speedup the search.

The group of Self-Contained Parallel Cooperation algorithms also known as the *Island model* is used for parallel systems with very limited communication. In the island model, every processor runs an independent optimization using a separate (sub)population. We study this family of the parallel models in the following.

The main topic of this chapter is about multi-objective optimization algorithms when employing them on a set of heterogeneous computing resources. The solution of multi-objective optimization problems is usually a set of solutions represented as an optimal front i.e., none of these solutions can be improved in one objective without getting worse with respect to some other objective. Some difficulties arise, though, when we solve multi-objective problems [23, 5, 10, 24, 2, 7].

Typically, solving multi-objective problems using a set of computing resources is achieved by dividing the task (finding the set of non-dominated solutions) between the computers. This idea has been studied e.g., in [5, 10]. The task partitioning can be successfully done, when there is a priori knowledge about the number of resources and their properties. In this case, if one of the resources fails, its related part has to be re-assigned. Also, the other issue is the assumption of communication between the processors like in [5].

In this chapter, we study how to partition the optimization task between the available processors meaning every computing resource is responsible for generating one part of the optimal front so that in a collective way all of them obtain a good approximation of the true Pareto-optimal solutions. We design the algorithm to be independent of the number of resources. The resources act as computing agents, which look for a partition to optimize and try to avoid overlaps. If one of them fails working, other resources which are done with their own tasks, take care of the missing partition. Besides this, we allow the cooperative aspect in the heterogeneous system such that the computing resources indirectly exchange the best found solutions. In this way, the computing resources are represented as a unified resource to the user, who gives the objectives to the system and receives the optimal solutions in the end whereas in the traditional parallel systems, the user has to assign the tasks and permanently monitors the system.

Another aspect here is the fact that in a heterogeneous system we must make use of all of the resources from very slow to very fast ones. In a non-parallel case all of the population members are available in each evaluation step. However in our new approach, the solutions of the fast resources build a disconnected front and continue the optimization where the slow resources must still finish their tasks. Indeed, this issue affects the results. If the fast processors wait for the slow ones, the results are changed as the solutions of a multi-objective problem depend on each other by the domination relation. This aspect is being studied on a test scenario of 100 heterogeneous resources where we add some waiting time to the fast processors and study the quality of the obtained solutions. Furthermore, we add some failures to the system and observe the quality of solutions in such unreliable environments.

The chapter is structured as follows. The next section is assigned to the basics in multi-objective optimization, and the model of self-contained parallel cooperation in optimization. In Section 3, we explain an approach called self-organized parallel cooperation and study different aspects in parallelization of multi-objective algorithms. Section 4 is dedicated to experiments and the last section concludes the chapter.

2.1 Multi-objective Optimization

A multi-objective optimization problem is of the form

$$\text{minimize} \quad \mathbf{f} = (f_1(\mathbf{x}), f_2(\mathbf{x}), \cdots, f_m(\mathbf{x})) \tag{1}$$

subject to $\mathbf{x} \in S$, involving $m(\geq 2)$ conflicting objective functions $f_i : \Re^n \rightarrow \Re$ that we want to optimize simultaneously. The parameters $\mathbf{x} = (x_1, x_2, \cdots, x_n)^T$ belong to the feasible region $S \subset \Re^n$.

We denote the image of the feasible region by $Z \subset \Re^m$ and call it a feasible objective region. The elements of Z are called objective vectors and they consist of objective (function) values $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \cdots, f_m(\mathbf{x}))$. A parameter vector $\mathbf{x}_1 \in S$ is said to *dominate* a parameter vector $\mathbf{x}_2 \in S$, iff (a) $\mathbf{x}_1$ is not worse than $\mathbf{x}_2$ in all objectives and (b) $\mathbf{x}_1$ is strictly better than $\mathbf{x}_2$ in at least one objective.

$\mathbf{x}_1 \in S$ is called *Pareto-optimal* if there does not exist another $\mathbf{x}_2 \in S$ that dominates it. Finally, an objective vector is called Pareto-optimal if the corresponding decision vector is Pareto-optimal. The main goal of multi-objective optimization algorithms is to approximate the set of Pareto-optimal solutions by a set of well-distributed solutions.

The importance of distributed computing is even greater for multi-objective than for single objective optimization. In case of having many objective functions, the number of solutions needed to represent the multi-dimensional Pareto-front is large and therefore we require a large population size to perform a good exploration in the parameter space.

2.2 *Self-contained Parallel Cooperation*

The model of self-contained parallel cooperation, also known as the island model, is quite adequate for solving multi-objective problems in parallel [23]. This model is mostly used, if there is a limitation in the communication between the processors. Here, we can parallelize the optimization so that every processor runs an independent optimization with the goal of covering only one part of the Pareto-front. The optimized regions are collected by a master resource and build the approximated Pareto-front. Figure 1 shows an example of an ideal partitioning of the Pareto-front into several partitions and assigning different parts as tasks to k processors. Every resource contains an optimizer and has to optimize the solutions in the allocated region. A problem in this approach is, though, that it is not clear initially how to

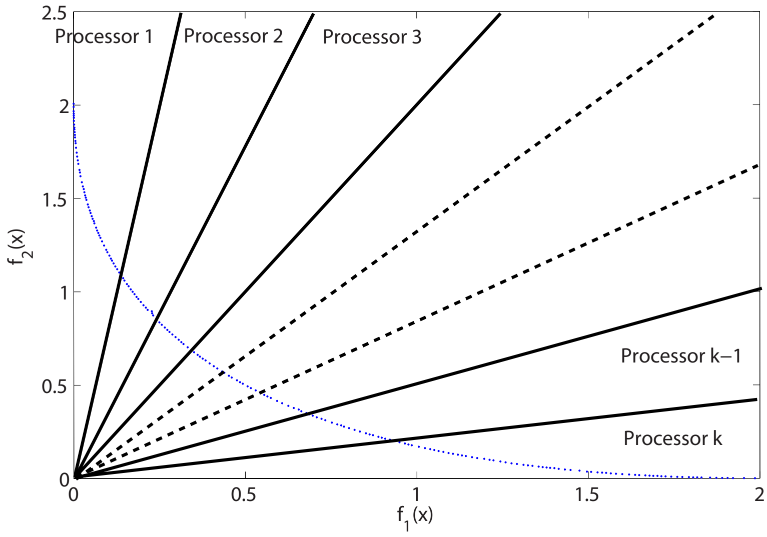


Fig. 1. An example of partitioning the objective space into sub-regions. The sub-regions are assigned to the k available resources.

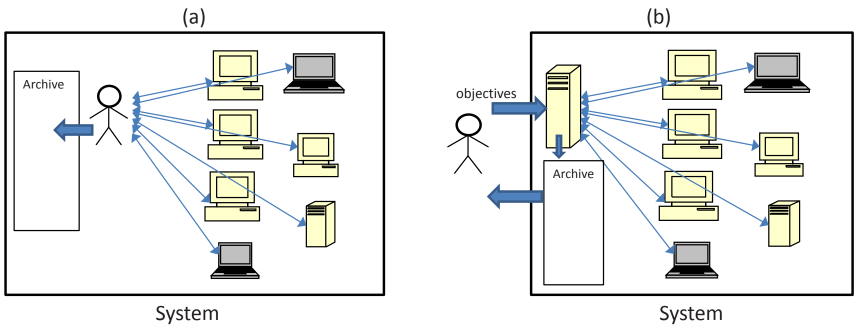


Fig. 2. (a) Traditional self-contained parallel cooperation model of optimization. The user is integrated into the system. (b) The self-organized parallel cooperation model, where the computing resources are represented as an unified resources to the user.

achieve a well-balanced distribution of the workload, since the solutions are not necessarily spread evenly over the Pareto-front.

This kind of optimization has been studied as a category of Self-Contained Parallel Cooperation [5, 10], where the user is aware of the exact number of available processors and divides the population of solutions into a fixed number of subpopulations. In such an approach, the user has to define the tasks of each processor, gather the obtained solutions from the resources, and build a global archive in which the non-dominated solutions are stored. Figure 2 (a) illustrates the system which optimizes the objective functions. The user is involved in the optimization process; in case that a processor fails working, the user manually reassigns the tasks.

3 Self-organized Parallel Cooperation

Based on the idea of self-contained parallel cooperation, we study a model containing a set of self-organized resources. This so called self-organized parallel cooperation is intended to change the system of computing resources into a unified resource to the user. Figure 2 (b) shows the main concept. The system contains the cooperating computing resources which are represented as a unified resources to the user. In contrast to the traditional parallel models (Figure 2 (a)), the user is not involved in the assignment of tasks to processors and is not aware of the number of resources and their corresponding properties such as speed. In such a system, the user defines the tasks (the objectives and the preferences) and obtains the optimal solutions after a certain time.

Generally, designing an algorithm for parallel implementation requires the following steps [8]: (1) Task partitioning which allocates independent tasks to multiple recourses (2) Task scheduling and assignment (3) Task synchronization which is about exchanging the information from the processors in order to ensure correct progress. These three aspects are considered in the self-organized model as follows:

- *Task partitioning*: The computing resources must find the most proper partition of the objective space to optimize. In traditional methods [5, 10], the space has been divided into a fixed number of sub-regions for a fixed number of resources. The resources consider their own sub-region as the feasible region and the solutions that are not in this region are considered to be infeasible. In this way, they force their populations to reside in the corresponding sub-region. In the self-organized parallel cooperation, this is completely different: The resources always look for the least populated part of the approximated front and start optimization in those areas (next section).
- *Task scheduling*: All of the resources from very fast to the slow ones have to collaborate to perform the optimization task. Also, in case of failures in one or more computing resources, the other resources take care of the failing subtasks.
- *Task synchronization*: There must be an indirect way of communication between the resources to ensure correct progress. The computing resources have to collect their results, remove the overlaps, and report the currently best solutions to the user.

In the following, we consider K heterogeneous computing resources which perform a multi-objective optimization algorithm for a specific range of parameters and a set of given objective functions. The resources may contain different algorithms and deliver a set of non-dominated solutions after a certain time T_i, $\forall i = 1, 2, \cdots, K$. We assume that the computing resources can communicate with a main master node in which we keep a global repository (called Archive) for storing the non-dominated solutions.

3.1 Task Partitioning

Depending on the optimization goals and the user preferences, every processor selects a sub-region of the objective or parameter space for optimization. For instance, the user preferences could be (a) a set of well-distributed solutions in the objective space or (b) a set of solutions with very good convergence to one predefined point or an area in the objective space or (c) both.

In the self-organized parallel cooperation model, the resources find the sub-regions based on the so far obtained solutions. For this purpose, as soon as a processor updates the global archive, it must evaluate the so far obtained non-dominated solutions in the archive. The evaluation of the archive members can be done by using a metric to observe the quality of the so far obtained solutions with respect to the user preferences.

3.1.1 Diversity and Convergence

Consider that a good distribution of solutions along the non-dominated front is the preference of the user. The resources must find the large gaps in the so-far-obtained front and concentrate on finding some optimal solutions in those regions. For this

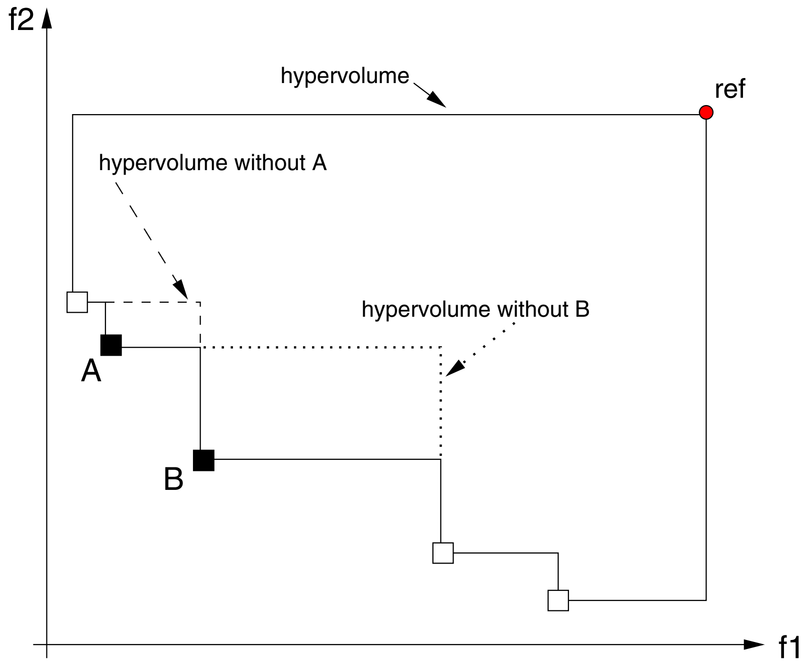


Fig. 3. Marginal Hypervolume Measure. The area between the non-dominated solutions and a predefined point called ref is shown by the solid line. The area (hypervolume) without A is smaller than the area without B indicating the importance of B.

purpose one proper metric is the marginal hypervolume measure. The hypervolume is the area dominated by all solutions stored in the archive [26]. The marginal hypervolume of a solution is the area dominated by the solution that is not dominated by any other solution. Figure 3 illustrates this. Solution A has a smaller contribution to the whole hyper-volume value than solution B. Therefore, solution B would be selected first. So the processor looking for a good sub-space can select the area around B to explore more in the next optimization run. B is called to be a reference point. In general as a reference point, the solution from the archive is selected which has not been selected before and which has the largest marginal hypervolume. In order to avoid that several computing resources select the same point, only if all archive solutions have been used as reference points, they are allowed to be re-used.

After selecting a proper reference point, the computing resource starts the optimization in that direction. The directed optimization can be performed in several ways such as by **reference based method** or **reference direction approach** in the context of non-evolutionary approaches [14] or by using the preference based multi-objective evolutionary approaches such as in [3]. We propose to use a guided “multi-objective particle swarm optimization” [17, 6, 20]. By giving the reference point as a global guide, the particles are drawn toward the reference point and hence the optimization concentrates around the desired area.

3.1.2 Multi-objective Particle Swarm Optimization

Algorithm 1 shows one typical structure of a Multi-Objective Particle Swarm Optimization (MOPSO). The Algorithm starts with a set of uniformly distributed random initial individuals (also called particles) defined in the search space S. A set of N particles are considered as a population P_t at the generation t. Each particle i has a position defined by $\mathbf{x}^i = (x_1^i, x_2^i, \cdots, x_n^i)$ and a velocity defined by $\mathbf{v}^i = (v_1^i, v_2^i, \cdots, v_n^i)$ in the parameter space S.

Algorithm 1. MOPSO Algorithm

```
Require: $N$
Ensure: $A$
  1. Initialization: Initialize population $P_t$, $t = 0$:
  for $i = 1$ to $N$ do
      Initialize $\mathbf{x}_t{}^i$, $\mathbf{v}_t{}^i = \mathbf{0}$ and $\mathbf{p}_t{}^i = \mathbf{x}_t{}^i$
  end for
  Initialize the archive $A_t := \{\}$
  2. Evaluate: $Evaluate(P_t)$
  3. Update: $A_{t+1} := Update(P_t, A_t)$
  4. Move: $P_{t+1} := Move(P_t, A_t)$
  for $i = 1$ to $N$ do
      $\mathbf{p}_t{}^{i,g} := FindBestGlobal(A_{t+1}, \mathbf{x}_t{}^i)$
      for $j = 1$ to $n$ do
          $v_{j,t+1}^i = wv_{j,t}^i + c_1R_1(p_{j,t}^i - x_{j,t}^i) + c_2R_2(p_{j,t}^{i,g} - x_{j,t}^i)$
          $x_{j,t+1}^i = x_{j,t}^i + v_{j,t+1}^i$
      end for
      if $\mathbf{x}_{t+1}^i \prec \mathbf{p}_t{}^i$ then
          $\mathbf{p}_{t+1}^i = \mathbf{x}_{t+1}^i$
      else
          $\mathbf{p}_{t+1}^i = \mathbf{p}_t{}^i$
      end if
  end for
  5. Termination: Unless a $termination\ criterion$ is met $t = t + 1$ and $goto$ Step 2
```

Beside the population, another set (called Archive) A_t can be defined in order to store the obtained non-dominated solutions. Due to the presence of an archive, good solutions are preserved during generations and therefore, convergence might be guaranteed [21]. In Step 2 of the Algorithm, the individuals are evaluated and the non-dominated solutions are added to the archive. Thereby, the archive is kept domination-free. Obviously, during the execution of the function *Update*, dominated solutions must be removed. This is done in Step 3 of the Algorithm 1.

In Step 4, the particles are moved to the new positions in the space. The velocity and position of each particle i is updated as below:

$$\begin{aligned} v^i_{j,t+1} &= wv^i_{j,t} + c_1R_1(p^i_{j,t} - x^i_{j,t}) + c_2R_2(p^{i,g}_{j,t} - x^i_{j,t}) \\ x^i_{j,t+1} &= x^i_{j,t} + v^i_{j,t+1} \end{aligned} \tag{2}$$

where $j = 1, \ldots, n$, $i = 1, \ldots, N$, c_1 and c_2 are two positive constants, R_1 and R_2 are random values in the range $[0, 1]$ and

- w is the so called **inertia weight** of the particle. This is employed to control the impact of the previous history of velocities on the current velocity, thus to influence the trade-off between global and local exploration abilities of the particles [22, 13]. A larger inertia weight w facilitates global exploration while a smaller inertia weight tends to facilitate local exploration to fine-tune the current search area. Suitable selection of the inertia weight w can provide a balance between global and local exploration abilities requiring fewer iterations for finding the optimum on average [22, 13]. A nonzero inertia weight introduces the preference for the particle to continue moving in the same direction as in the previous iteration.
- c_1R_1 and c_2R_2 are called **control parameters** [13]. These two control parameters determine the type of trajectory the particle travels. If R_1 and R_2 are 0.0, it is obvious that $v = v + 0$ and $x = x + v$ (for $w = 1$). It means the particles move linearly. If they are set to very small values, the trajectory of x rises and falls slowly over time.
- $\mathbf{p}_t^{i,g}$ is the position of the **global best particle** in the population, which guides the particles to move toward the optimum. The important part in MOPSO is to determine the best global particle $\mathbf{p}_t^{i,g}$ for each particle i of the population. In single-objective PSO, the global best particle is determined easily by selecting the particle that has the best position. But in MOPSO, $\mathbf{p}_t^{i,g}$ must be selected from the updated set of non-dominated solutions stored in the archive A_{t+1}. Selecting the best local guide is achieved in the function $FindBestGlobal(A_{t+1}, \mathbf{x}_t^i)$ for each particle i [17].
- $\mathbf{p}_t^i$ is the best position that particle i could find so far [4]. This is like a memory for the particle i and keeps the non-dominated (best) position of the particle by comparing the new position $\mathbf{x}_{t+1}^i$ in the objective space with $\mathbf{p}_t^i$ ($\mathbf{p}_t^i$ is the last non-dominated (best) position of the particle i).

The steps of a MOPSO are iteratively repeated until a termination criterion is met, such as a maximum number of generations, or when there has been no change in the set of non-dominated solutions for a given number of generations. The output of an elitist MOPSO method is the set of non-dominated solutions stored in the final archive.

In MOPSO, we also define a parameter called **turbulence factor** which is basically designed to avoid the local optima. With a probability value equal to the turbulence factor, a particle is moved to a random position in the search space. It is obvious that if we increase the turbulence factor, the number of random solutions increases.

One advantage of MOPSO is that the particles move in the parameter space. This can be used to easily guide the particles toward a preferred area by selecting the preference point as the global best particle for all the particles in the population [25, 18]. This is called Guided MOPSO. Guided MOPSO can not only be used for solving preference based optimization problems but can highly be useful in solving Many-Objective Optimization problems [19].

3.2 Task Scheduling

As soon as a resource finds the reference point, it starts the optimization. The scheduling is therefore straight forward. Algorithm 2 shows the routine performed in each of the resources. In the case that the entire system can be used for a certain time $T_{total}(\geq T_i)$, $(i = 1, 2, \cdots, K)$ fast resources might be able to perform several optimization tasks depending on their computation power. In this algorithm, whenever a processor is done with the optimization, the obtained results are integrated into the (global) archive in which only non-dominated solutions are kept.

Algorithm 2. Task of resource i

```
repeat
    Find a task partition
    Perform the optimization
    Send update to main archive
until ($T_i \leq T_{total}$)
```

3.3 Task Synchronization

The indirect communication between the resources is indeed achieved via the archive. Every computing resource must select a task to optimize and continue the optimization as long as the total computation time is not over.

Note that the above approach studied in Sections 3.1–3.3 reduces the waiting time of the traditional self-contained parallel cooperation model in which the fast resources do not wait for the slow ones. Indeed, saving computation time is a desirable fact, but note that the solutions of a multi-objective problem depend on each other. This means that if the fast processors wait for the slow ones, the quality of solutions might get better. This will be studied in the experiments (Section 4). In this approach, any desirable interaction between a decision maker and the system can be easily implemented. Also the communication overhead between the resources is reduced to the communication between the resources and a master node managing the archive which could be viewed as some type of blackboard system. Except for the master node, failures are automatically compensated, because other processors will draw their attention to a region that has not been worked on. Obviously, though, a failure of the master node would require a restart of the optimization process.

4 Experiments

The major goal of the experiments is to investigate the influence of failures and waiting time on the results of a multi-objective optimization method when using a heterogeneous environment. Therefore, we only select one standard test function and observe the quality of solutions when running the system for a certain time and adding failures to the resources. The system is also analyzed by adding waiting time to the resources to wait for a certain percentage of the others. The experiments are performed on a simulation environment containing 100 resources with 5 different computation speeds (types). The simulation is based on the real scenario in a typical Grid. There are 3, 3, 20, 43, and 31 number of type 1 (very fast), type 2, type 3, type 4, and type 5 (very slow) resources, respectively.

The test function selected from [11, 9] contains 10 parameters and 2 objectives:

$$f_1(\mathbf{x}) = 1 - exp(-\sum_{i=1}^{n}(x_i - \frac{1}{\sqrt{n}})^2)$$
$$f_2(\mathbf{x}) = 1 - exp(-\sum_{i=1}^{n}(x_i + \frac{1}{\sqrt{n}})^2)$$

where $x_i \in [-4,\ 4]$ and $n = 10$. The quality of the solutions is computed by the hypervolume metric [26] averaged over 20 runs.

4.1 Waiting Time

The first experiments are intended to analyze the behavior of the system when adding waiting time to the processors. Therefore, the following parameter setting is proposed. Every resource contains a multi-objective particle swarm optimization from [17] with one particle running for one generation. The total time T_{total} is set to be 120, meaning the fast resources are able to evaluate 120 particles. The given user goals besides the objective functions are to achieve a good convergence and spread of solutions. For this, we employ the idea of using marginal hypervolume presented in the example in Section 3.1.

The main archive stored in the master node is set to be empty at the beginning (first run). The selected parameter settings indicate that the first runs of the 100 resources include only a random sampling of the search space until the fast processors finish the evaluations and update the archive. As soon as the archive is updated, the preference of the user can be considered.

Figure 4 shows the quality of the archive members over time. Different plots illustrate the quality if the done resources wait for j other resources to finish and update the archive ($j = 0, 10, 20, 30, 40, 50$).

We observe that if the resources do not wait, the results are worse than if they wait for at least 10 to 20 percent of the resources. This is an expected result as in multi-objective optimization the results depend on each other through the dominance relation. If the resources wait for (in this case) up to 20 percent, they achieve

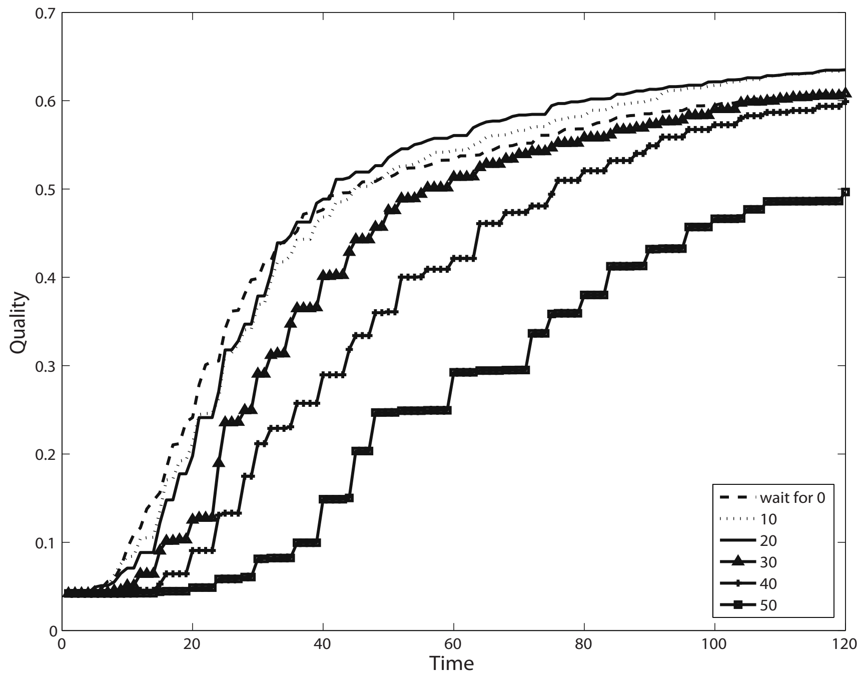


Fig. 4. The quality of the obtained solutions over time. The plots show the quality if the computing resources wait for at least 0, 10, 20, 30, 40 and 50 other computing resources to continue the optimization. A long waiting time indicates the worst quality, whereas waiting time for less than 20 percent is shown to increase the quality.

a better quality than if they do not wait. In fact, in every step of optimization it is better to have enough evaluations to find more dominated and non-dominated solutions. Hence the waiting time has the advantage of having more solutions. This leads to a better direction in the optimization than not waiting. However, waiting for a large number of other resources means that the fast processor stay for a long time in idle mode which is on the other hand not desirable when having a fixed time for the entire optimization process (like in the Grid).

4.2 *Unreliable Environment*

The next experiments are dedicated to study failures in the system. Here we also analyze the waiting time. The failures are simulated by randomly removing some number of resources from the system. Figure 5 shows the quality of the archive members for different failure rates. The failure rates change from 0 to 35 percent. The result of having zero failure rate is the same as the results in Figure 4. Here we observe if we increase the failure rate, the best results are obtained by zero waiting

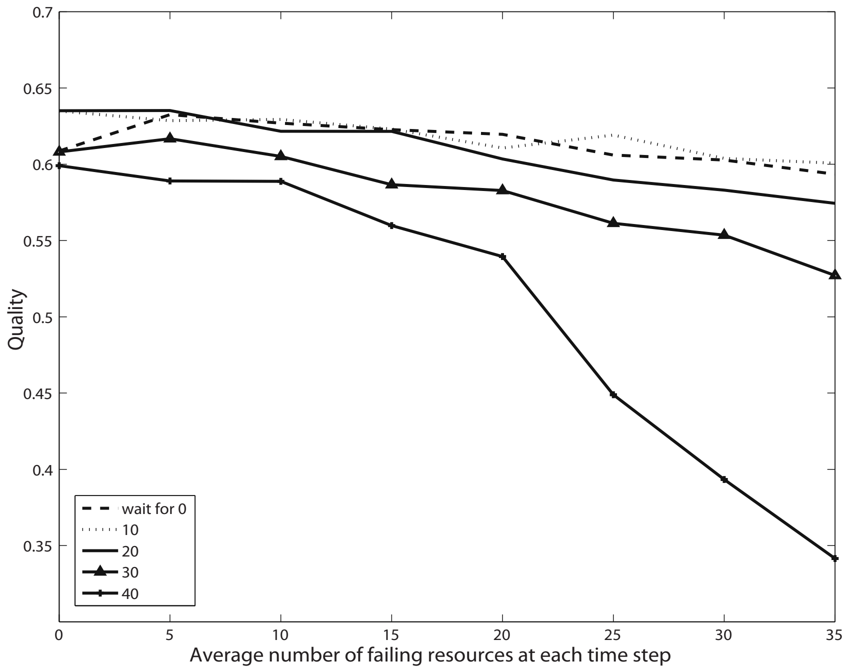


Fig. 5. The quality after the total time of 120 for different failure rates. The different plots indicate the waiting time for the resources.

time. This indicates that in unreliable systems, the fast resources have to make use of the available computing time and do not wait for the slow resources, where in a reliable environment waiting for up to 20 percent of the slower resources leads to a better quality of the results.

5 Conclusions

In this chapter, we study a parallel model called self-organized parallel cooperation for solving multi-objective optimization problems using a set of heterogeneous computing resources. Apart from gaining computation time through the parallelization, the quality of the solutions can be improved by this approach. The computing resources (computing agents) run optimization algorithms to solve different parts of the approximated front in the way that they observe the so far obtained quality of solutions over time and select a sub space to optimize.

This approach has been studied on a scenario containing 100 computing resources with 5 different speeds. The scenario is depicted from a real grid environment with different failure percentage of computing resources. Furthermore, we analyze the quality of solutions, when keeping the fast processors to wait for the slow ones for a certain time. The experiments illustrate that when dealing with reliable environments, i.e., here no failure rates, the best solutions are obtained when the fast processors wait for some of the slow ones. However in unreliable environments, where some resources randomly fail during the run time, waiting does not result in the best quality.

This approach can be studied in the context of interactive optimization methods in which the user changes the preferences during the optimization. However, other failure models as studied in this chapter and several other aspects in unreliable environments must be further investigated.

References

1. Abramson, D., Lewis, A., Peachy, T.: Nimrod/o: A tool for automatic design optimization. In: The 4th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP 2000 (2000)
2. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation 6(5), 443–461 (2002)
3. Branke, J., Deb, K., Miettinen, K., Slowinski, R.: Multiobjective Optimization Interactive and Evolutionary Approaches. Springer, Heidelberg (2008)
4. Branke, J., Mostaghim, S.: About selecting the personal best in multi-objective particle swarm optimization. In: Runarsson, T.P., et al. (eds.) Parallel Problem Solving from Nature. LNCS, pp. 523–532. Springer, Heidelberg (2006)
5. Branke, J., Schmeck, H., Deb, K., Reddy, M.: Parallelizing Multi-Objective Evolutionary Algorithms: Cone Separation. In: IEEE Congress on Evolutionary Computation, pp. 1952–1957 (2004)
6. Bui, L.T., Abbass, H.A., Essam, D.: Local models - an approach to distributed multiobjective optimization. Journal of Computational Optimization and Applications (2007)
7. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer, Dordrecht (2000)
8. Censor, Y., Zenios, S.A.: Parallel Optimization: Theory, Algorithms, and Applications. Oxford University Press, Oxford (1997)
9. Coello Coello, C.A., Van Veldhuizen, D.A., Lamont, G.B.: Evolutionary Algorithms for Solving Multi-Objective Problems. Kluwer Academic Publishers, Dordrecht (2002)
10. Deb, K., Zope, P., Jain, A.: Distributed computing of pareto-optimal solutions with evolutionary algorithms. In: Fonseca, C.M., Fleming, P.J., Zitzler, E., Deb, K., Thiele, L. (eds.) EMO 2003. LNCS, vol. 2632, pp. 534–549. Springer, Heidelberg (2003)
11. Fonseca, C.M., Fleming, P.J.: On the Performance Assessment and Comparison of Stochastic Multiobjective Optimizers. In: Ebeling, W., Rechenberg, I., Voigt, H.-M., Schwefel, H.-P. (eds.) PPSN 1996. LNCS, vol. 1141, pp. 584–593. Springer, Heidelberg (1996)
12. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Publishing Company, Inc., Reading (1989)
13. Kennedy, J., Eberhart, R.C.: Swarm Intelligence. Morgan Kaufmann, San Francisco (2001)
14. Miettinen, K.M.: Nonlinear Multiobjective Optimization. Kluwer Academic Publishers, Dordrecht (1999)

15. Mostaghim, S., Branke, J., Lewis, A., Schmeck, H.: Parallel multi-objective optimization using a master-slave model on heterogeneous resources. In: IEEE Congress on Evolutionary Computation (2008)
16. Mostaghim, S., Branke, J., Schmeck, H.: Multi-objective particle swarm optimization on computer grids. In: The Genetic and Evolutionary Computation Conference, vol. 1, pp. 869–875 (2007)
17. Mostaghim, S., Teich, J.: Strategies for finding good local guides in multi-objective particle swarm optimization. In: IEEE Swarm Intelligence Symposium, pp. 26–33 (2003)
18. Mostaghim, S., Teich, J.: Covering pareto-optimal fronts by subswarms in multi-objective particle swarm optimization. In: IEEE Proceedings, World Congress on Computational Intelligence (CEC 2004), Portland, USA (June 2004)
19. Mostaghim, S., Schmeck, H.: Distance based ranking in many-objective particle swarm optimization. In: Rudolph, G., Jansen, T., Lucas, S., Poloni, C., Beume, N. (eds.) PPSN 2008. LNCS, vol. 5199, pp. 753–762. Springer, Heidelberg (2008)
20. Reyes-Sierra, M., Coello, C.A.C.: Multi-objective particle swarm optimizers: A survey of the state-of-the-art. International Journal of Computational Intelligence Research 2(3), 287–308 (2006)
21. Rudolph, G., Agapie, A.: Convergence Properties of Some Multi-Objective Evolutionary Algorithms. In: Proceedings of the 2000 Congress on Evolutionary Computation, pp. 1010–1016 (2000)
22. Shi, Y., Eberhart, R.C.: Parameter selection in particle swarm optimization. Evolutionary Programming, 591–600 (1998)
23. Talbi, E.-G., Mostaghim, S., Okabe, T., Ichibushi, H., Rudolph, G., Coello Coello, C.: Parallel Approaches for Multiobjective Optimization, pp. 349–372. Springer, Heidelberg (2008)
24. Van Veldhuizen, D.A., Zydallis, J.B., Lamont, G.B.: Considerations in engineering parallel multiobjective evolutionary algorithms. IEEE Transactions on Evolutionary Computation 7(2), 144–173 (2003)
25. Wickramasinghe, U.K., Li, X.: Integrating user preferences with particle swarms for multi-objective optimization. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 745–752 (2008)
26. Zitzler, E.: Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. Shaker (1999)

The Role of Explicit Niching and Communication Messages in Distributed Evolutionary Multi-objective Optimization

Lam T. Bui, Daryl Essam, and Hussein A. Abbass

Abstract. This chapter is dedicated to an investigation on the role of explicit niching and communication messages in distributed evolutionary multi-objective optimization. Localization is employed to implement explicit niching. Several options are selected for communication messages including non-dominated solutions and statistics such as the centroid of the non-dominated set, the direction of improvement, or weighted direction of improvement. As a result, a distributed system using the framework of local models is developed to support distributed computing in evolutionary multi-objective optimization. This system provides a flexibility in applying different architectures such as master/slave, island as well as the hybridization of the two. An in-depth analysis is carried out on a simulation study using the system.

1 Introduction

There have been a significant number of contributions to the area of distributed evolutionary multi-objective optimization [14, 10, 29, 3, 25, 32, 26, 27]. This fact can be seen partially as a result of a radical revolution in hardware technology for distributed computing; for example, clustering systems using switches with Gigabit-speed have become commonly affordable and therefore been popular worldwide. For the domain of multiobjective evolutionary algorithms (MOEAs), communication (exchanging/migrating of solutions) is more difficult than that in the single-objective domain. Exchanging individuals among nodes in the single-objective domain is quite deterministic [6], for example

Lam T. Bui · Daryl Essam · Hussein A. Abbass
The Artificial Life and Adaptive Robotics Laboratory (ALAR),
School of Information Technology and Electrical Engineering,
University of New South Wales at Australian Defence Force Academy,
Canberra, 2600, Australia

exchanging one or more best solutions; while in the multi-objective domain, it is not easy to determine the best solutions since there exists a set of trade-off good solutions. If all solutions of this set are sent, the communication cost is likely to become high. If too few solutions are sent, the information might not be enough to represent the set.

To date, the majority of the work in the multi-objective domain have been focusing on distributed architectures and communication topologies. A considerably small discussion has been dedicated to the role of communication messages (information content), an important factor that directly affects the communication cost of systems, together with parallel niching in distributed computing. Parallel niching plays an important role in a distributed system. It requires communication among nodes with communication messages containing niching information.

This chapter argues that communication messages are a major factor affecting the communication cost of a distributed MOEAs system, while explicit niching considerably contributes to the performance of the distributed system. In supporting this hypothesis, we employ the Evolutionary Multi-objective Optimization (EMO) framework using localization that was primarily proposed and used in [5, 4]. There, each sub-area of the search space is associated with a separate population and is used to build a local model. These models are allowed to interact using Particle Swarm Optimization (PSO)-inspired rules (or simply PSO rules [11]) to exchange local/global best information; and therefore, niching is somewhat maintained over time.

The main contribution of this chapter is a finding on the role of communication messages and explicit niching in distributed population-based multi-objective optimization. The communication cost can be reduced by using summary information of the system's progress instead of the solutions themselves, while maintaining the performance of the systems. The summary information on the optimal solutions can be used either partially or entirely as the content of the communication messages. Further, explicit niching is an important factor in obtaining the summary information.

A further contribution of the chapter is a flexible distributed framework of using local models to solve multi-objective optimization problems (MOPs). Although the approach of building local models based on localization has been used elsewhere in the literature, the novelty of this work is the treatment of each local model as a single particle and the use of PSO rules to control these particles. The explicit niching is maintained over time by these rules. This provides a control mechanism over the balance between exploration and exploitation. The system using this framework was shown to offer a quick convergence to the set of optimal solutions. A theoretical paradigm of computational time for the framework was developed.

The remainder of the chapter is organized as follows: All the background information on distributed MOEAs is given in the second section. A short description is given in Section 3 for the framework of local models. A design for the proposed system is described in Section 4. An investigation on the role of communication in distributed systems is given in Section 5, and the last section presents conclusions.

2 Distributed Evolutionary Multi-objective Optimization

In general, distributed MOEAs are classified into four broad streams using different architectures: Master-slave, Island, Diffusion and Hybrid models [29]. Master-slave models are the simplest form of distributed MOEAs. A processing node is set to be a master. This node controls the entire evolutionary process in which it distributes objective evaluations to all slave nodes and decides how to deal with these objective values. This approach does not change the course of action of the optimization process, but it helps to reduce the processing time by distributing it on several processing nodes. Several instances of master/slave can be found in [24, 17, 16, 18, 20, 31, 32, 26].

Island models (sometimes called coarse-grained models) are inspired by natural phenomena [21, 7, 2, 30, 31, 13, 23, 26, 12]. The population is constructed from a set of sub-populations. Each subpopulation, called an island, runs on a processing node with their own MOEA. The frequency of communication among islands is also defined. The islands cooperate with each other by exchanging good individuals in order to help other islands avoid loss of diversity and to simultaneously progress towards the global Pareto optimal front (POF). This task is usually called migration. It usually follows a predefined topology.

Diffusion models (or fine-grained models) also use a conceptual population during the optimization process as in master/slave models. However, each processor only deals with few individuals [22, 29, 19]. Further, their topology is usually defined logically as a grid. The neighborhoods are then defined by a number of possibly overlapping rectangles, whereby good individuals can diffuse through these overlapping regions. One interesting point is that the model allows regions to overlap and to also sometimes be merged into a larger region. This could make the model more elegant when approaching the global POF, but it might be costly due to the high communication load of the system.

It is a fact that the master/slave architecture initially attracted a lot of attention from the research community. That might be because of the relative easiness in its implementation. However, the island architecture has recently gained the lead. Note that the island architecture offers a flexibility in controlling the course of action via interaction among nodes. Meanwhile, diffusion and hybrid architectures has had less published work probably because their implementations are much more complicated than the other two and they usually (especially diffusion) requires some special support of hardware that might not always be available in practice.

Communication is a vital part of the distributed system. However, implementation of communication is dependent on the particular architecture. For examples, for the master/slave architecture, communication occurs between the master and all other slaves (no communication among slaves), while for island models, communication can happen between any two nodes. In general, for communication, it is important to address the issues of logical topology (defining neighborhood relationship) [29], communication messages (exchanging information among processing nodes), and frequency (intervals of exchanging) [6, 10, 27].

It is also a fact that there has been very little discussion on the use of summary information for communication messages; for example, in [33], the authors proposed to use information of the shapes of the non-dominated sets in the objective space as the communication messages. However, there was a lack of depth and systematic analysis on the role of communication messages regarding the use of different kinds of information as communication messages.

Niching is a popular topic in both single and multi-objective evolutionary optimization. For MOEAs, niching can be done either in the decision or objective space. Although the matter of niching in MOEAs has become mature with a plethora of techniques, it has not yet been deliberately discussed for distributed MOEAs (especially regarding the communication cost) [29].

Parallel niching is an architecture-dependent process. In general, parallel niching can be done in two ways: (1) Explicit division ('divide and conquer') of the search space, where each population is allowed to track a part of the search space over time. It is refereed to as *explicit niching* [14, 10, 28, 3, 25, 12, 15, 27]; and (2) obtaining niching information such as niche count and communicate them during the course of action. Populations adjust themselves according to these information [33, 29]. It is referred to as *implicit niching.*

3 Localization-Based Distributed Evolutionary Multi-objective Optimization System

Here, we employ the EMO framework using localization that was primarily proposed and used in [5, 4]. Each sub-area of the search space is associated with a separate population and is used to build a local model. These models are allowed to interact using PSO-inspired strategies (RACING, PSOV1, and PSOV2) to exchange local/global best information.

Our focus is how to build a distributed system based on the framework of local models with communication. For the system using this framework, it is called Localization-based Distributed Evolutionary Multi-objective Optimization System (LOS-EMOS), and the strategy of PSOV1 is selected when running experiments on this system.

Using the local models, it is expected that computation can easily be parallelized by running each model on at least one processing node. Note that there are two types of parallel processing: synchronous and asynchronous. However, this chapter focuses only on the synchronous type since it has been widely discussed in the literature and is more predictable and replicable than its asynchronous counterpart. Further, it is particularly more suitable for the proposed framework of local models.

3.1 Distributed Architectures

First of all, as indicated above, the original communication channel in the system using the framework of local models suggests that when applied to distributed computing, *the architecture of master/slave* (or shortly the master/slave) can be used for LOD-EMOS in the first place. For this architecture:

- Each sub-population (for a local model) is located on a node (slave).
- The global archive will be managed by a central node (master). This master will be in charge of coordinating all nodes. To reduce the idle time, we designate a slave to work as the master.

However, it differs from the conventional master/slave architecture in the way nodes communicate with the master. For example, each node sends its own non-dominated set to the master for archiving. The master is in charge of combining all received solutions to build the global archive. It then sends back to all nodes only the summary information: the *centroid of the global archive*. Further, the execution is distributed over all nodes.

Meanwhile, in the conventional master/slave, the master sends all individuals to nodes for evaluation and then receives back all objective values. The execution is centralized at the master and all slaves are just for individual evaluations. In general, the master/slave is an effective one, but it can result in the bottleneck effect from the master node in some cases of slow networks.

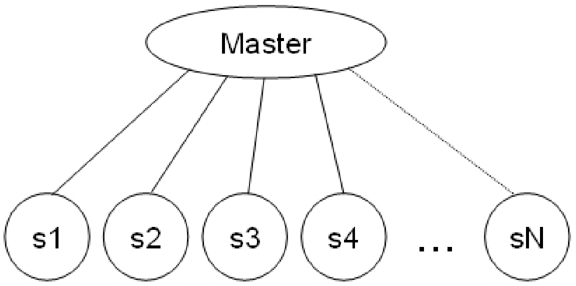


Fig. 1. An example of master/slave architecture

Another alternative to the master/slave in the design of a distributed architecture using local models is *the island architecture*, where nodes directly communicate with their neighbors. In this design, the neighborhood of nodes is defined by using ring topology. However, in order to fit local models into the island architecture, it needs to address the matter of the global centroid. Since all nodes communicate following the ring, it is not necessary to have a global centroid for all nodes, instead the global centroid should be used for the node of interest and its neighbors only. One example is that for each node in the ring, its global centroid will be the centroid of the local archive of its previous node; and the communication rules are still applicable. Therefore, the communication here might be the matter of sending and receiving the centroids of all private archives of the sub-populations. For the issue of data transferring, it reduces very much the amount of transferred data in comparison with the master/slave.

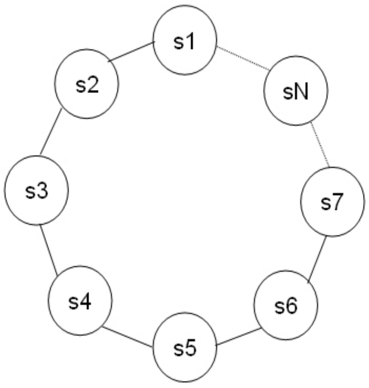


Fig. 2. An example of the island architecture

The last architecture is hybridization of the both above ones where there are several masters communicated with ring topology. All slaves are broken down into several groups (clusters); each cluster will be controlled by one master. Further, a master is selected to coordinate all masters. This is called *clustering architecture*. With this architecture, each sub-population will be located on one slave node. Each master will maintain a global archive for one cluster. Therefore, the master will receive information from all nodes of its cluster and from its previous master in the ring. It seems to increase the workload of data transferring. However, the use of multiple masters will help to reduce the bottleneck effect of the master/slave architecture with one master.

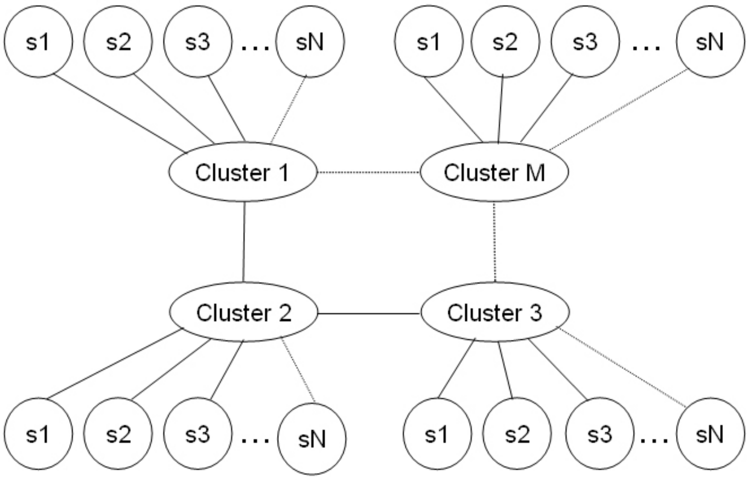


Fig. 3. An example of the clustering architecture

3.2 Communication Messages

Communication messages Type 1- CM1: This type of information is used to adapt with the original design of the local models where the local non-dominated solutions are used to build the global archive for the system guidance. Therefore, local non-dominated solutions from nodes will be the objects to be transmitted on the communication channel. However, the actual transmission will depend on the particular architecture:

– **Master/slave:** With this architecture, the master receives all non-dominated solutions from all slaves. Note that each solution contains an array of decision variables and another array of objective values. In turn, it sends back to slaves summary information only: an array of the global centroid. In term of the size, this array is equal to the array of decision variables. Here, it is assumed that the data for each solution is a record of:

  - An array of decision variables. Also assume that the data type for decision variables are double (it is a 8-byte data type)
  - An array of 8-byte objective values

  Further, for a centroid, its size is equal to the size of an array of decision variables. Therefore, the size of a solution and a centroid will be $(n + m) \times 8$ and $n \times 8$ respectively, where n is the problem dimension, m is the number of objectives. Overall, the amount of data transferred during the optimization process is determined as follows:

$$A = \sum_{j=1;j=j+f}^{G} \left[\sum_{i=1}^{P-1} [N_i^j \times (n+m) \times 8] + (P-1) \times n \times 8 \right] \quad (1)$$

where G is the number of generations, N_i is the number of non-dominated solutions in the slave i, f is the communication frequency, and P is the number of nodes. In the worst case, the number of non-dominated solutions is equal to the slave's sub-population size; or $N_i = \frac{N}{P}$, N is the total population size

– **Island:** As indicated in the previous sub-section, each node in the island architecture has its own global centroid. It needs information of the non-dominated set of its previous node in the ring to determine the global centroid. Sine there is no need to combine the non-dominated solutions from two nodes, the communication messages can be just only the centroid information. Each node sends its own centroid to the other, while receiving the other's centroid. This will help to reduce a huge amount of data transferred in one generation. Therefore, the amount of transferred data is as follows:

$$A = \left[a + \frac{G}{f}\right] \times P \times n \times 8 \tag{2}$$

where:

$$a = \begin{cases} 1 \text{ if } f > 1 \\ 0 \text{ if } f = 1 \end{cases} \tag{3}$$

– **Clustering:** In this case, there are serval masters connected via a ring. Each master is in charge of a cluster (with a number of slaves). Therefore, the communication includes sending non-dominated solutions from slaves to masters, from master to master, and centroids from master to slaves. Eq. 4 is used to calculate the amount of transferred data

$$\begin{aligned} A = \textstyle\sum_{j=1;j=j+f}^{G} \Big[& \textstyle\sum_{i=1}^{P-M} [N_i^j \times (n+m) \times 8 + \\ & + n \times 8] + \textstyle\sum_{i=1}^{M} [C_i^j \times (n+m) \times 8] \Big] \end{aligned} \tag{4}$$

where M is the number of clusters and C_i is the number of non-dominated solutions in the archive of the master i.

Communication messages Type 2 - CM2: For this type of communication, LOD-EMOS will exploit another type of summary information: the direction of improvement. For each local model, the direction of improvement is essential for its progress to the POF; and obviously the combination of the directions from all models gives rise to the global direction of the system. If this global direction is used to determine the global centroid for the next time step, the final effect becomes quite the same as the way of using non-dominated solutions to define the global centroid. This can be considered as the reduced form of the original method of computing the global archive. However, in some sense, it still reserves information of the global direction of improvement. Furthermore, if looking at the aspect of distributed computing, it reduces significantly the amount of transferring data. In the following, the effect of using direction on different architectures will be discussed:

- **Master/slave:** Again the slaves need to send the master information about their progress (i.e. the obtained non-dominated sets) and then the master will send back the global centroid. However, what they send to the master is only the direction of improvement. Note that it has the equal size to the size of a centroid. The global direction $\overrightarrow{D^{t+1}}$ and centroid $\overrightarrow{c_g^{t+1}}$ will be calculated as follows:

$$\overrightarrow{D^{t+1}} = \sum_{i=1}^{P} \overrightarrow{d_i^t} \tag{5}$$

$$\overrightarrow{c_g^{t+1}} = \overrightarrow{c_g^t} + \overrightarrow{D^{t+1}} \tag{6}$$

  Note that in the initial generation, all non-dominated sets are sent to determine the initial global centroid. Hence, the amount of data will be as follows:

$$A = A_0 + \frac{G}{f} \times [2(P-1) \times n \times 8] \tag{7}$$

  $n \times 8$ is the size of transferred data for one record of direction or a centroid; A_0 is the amount of transferred data of the initial generation and

$$A_0 = \left[\sum_{i=1}^{P-1} [N_i^0 \times (n+m) \times 8] + (P-1) \times n \times 8 \right] \tag{8}$$

- **Island:** In this case, there is no difference between using direction or non-direction to communicate. Sending a direction or a centroid to other node causes the same amount of data (see Eq. 2). Further, the final global centroid of a node is the same for both cases. Therefore, the performance is exactly the same. The global direction for each slave i is now determined as in Eq. 9

$$\overrightarrow{D_i^{t+1}} = \overrightarrow{d_{neighbor}^t} \tag{9}$$

- **Clustering:** Similar to the master/slave architecture above, the slaves or masters communicate by using direction instead of non-dominated solutions.

  The global direction of each cluster k is calculated as follows:

$$\overrightarrow{D_k^{t+1}} = \sum_{i=1}^{P1} \overrightarrow{d_i^t} + \overrightarrow{dc_{neighbor}^t} \tag{10}$$

  where P1 is the number of nodes associated with a master, and the amount of transferred data:

$$A = A_0 + \frac{G}{f} \times [(2P - M) \times n \times 8] \tag{11}$$

$$A_0 = \left[\sum_{i=1}^{P-M} [N_i^0 \times (n+m) \times 8 + n \times 8] + \sum_{i=1}^{M} [C_i^0 \times (n+m) \times 8] \right] \tag{12}$$

Communication messages Type 3 - CM3: This is an extension of the second type. It still uses the direction of improvement. However, nodes need to send also the size of their non-dominated sets (assumed *2-byte integer*). This value will be used to weight the contribution of its local direction to the global direction. It will be useful in some situations such as it gives nodes, which are in good areas but make only a little move, more influence in the global direction instead of being forgotten. This also somewhat compensates for the loss of information that the use of direction can cause in comparison with CM1. The differences in applying to different architectures are highlighted as below:

– **Master/slave:** For this architecture, the global direction is calculated as follows:

$$\overrightarrow{D^{t+1}} = \sum_{i=1}^{P} \frac{N_i}{\sum_{j=1}^{P} N_j} \overrightarrow{d_i^t} \tag{13}$$

Also, the amount of transferring data is calculated as in Eq. 14

$$A = A_0 + \frac{G}{f} \times [2 \times (P-1) \times n \times 8 + (P-1) \times 2] \tag{14}$$

– **Island:** For an arbitrary node i, its global direction is calculated by combining its neighbor's direction and itself. The global direction is as the following:

$$\overrightarrow{D_k^{t+1}} = \left[\frac{N_i}{N_i + N_{neighbor}} \overrightarrow{d_i^t} + \frac{N_{neighbor}}{N_i + N_{neighbor}} \overrightarrow{d_{neighbor}^t} \right] \tag{15}$$

Also, the amount of transferred data is:

$$A = \left[a + \frac{G}{f} \right] \times (P \times n \times 8 + P \times 2) \tag{16}$$

a is calculated as in Eq. 3

– **Clustering:** In this case, it is necessary to calculate global direction from M masters.

$$\begin{aligned} \overrightarrow{D_k^{t+1}} &= \textstyle\sum_{i=1}^{P1} \frac{N_i}{\sum_{j=1}^{P1} N_j} \overrightarrow{d_i^t} + \left[\frac{NC_i}{NC_i + NC_{neighbor}} \overrightarrow{dc_i^t} \right. \\ &\quad \left. + \frac{NC_{neighbor}}{NC_i + NC_{neighbor}} \overrightarrow{dc_{neighbor}^t} \right] \end{aligned} \tag{17}$$

The amount of transferred data:

$$A = A_0 + \frac{G}{f}\times[(2P - M)\times n\times 8 + P\times 2] \tag{18}$$

3.3 Communication Frequency

The frequency of communication is used to determine the time for nodes to communicate with the masters or other nodes. For LOD-EMOS (or other distributed systems in general), the frequency is very important since the nodes rely very much on the globally guided information that is communicated among nodes. A high frequency might result in premature convergence since the global information influences too much on the local models and there is little time for them to conduct their own exploration. The local models would thus quickly jump into the same areas before finding the true POF. On the other hand, if the frequency is too low, the effect of using global information to guide the search will be limited.

In general, the question of a suitable frequency is still open. Several works have limited theoretical discussions in only two cases: no communication and communication for every generation [6]. The common empirical choice is to use a fixed and small frequency. It primarily is also used for experiments in this chapter.

3.4 Time Analysis

Analysis of the time is essential for a distributed system since it is directly related to the efficiency of the system in comparison with its serial counterpart. In general, the computation time of a distributed system includes the evaluation time, execution time, and the communication time. As pointed out in [29], researchers ignore the execution time since it is usually much less than evaluation time and communication time. Also, it is usually the case that the execution time on serial and parallel is the same. The extra time, that the parallel system brings in, is the communication time. The parallel speedup has the meaning only in the case that this time is much less than the evaluation time.

As analyzed before, the framework of local models does not impose an extra computational complexity on the selected MOEA. Therefore, the execution time of the system using the local models is dependent only on the complexity of the selected MOEA. For example, if NSGA-II is employed, the execution time is the combination of the time for crossover, mutation, selection, non-dominated sorting for the local archive and global archive (if necessary). In this chapter, since the main focus in on the communication, the execution time is ignored.

The total computation time T is calculated as follows:

$$T = T_f + T_c \tag{19}$$

T_f is the evaluation time and T_c is the communication time. In the serial mode $T_c = 0$. Assuming that the environment is homogenous, then

$$T_f = \frac{GNt_f}{P} \tag{20}$$

where t_f is the time to evaluate one individual and N is the population size. Note that for heterogenous ones: $T_f = G * max(\frac{Nt_f^i}{P})$

The communication time is estimated as follows:

$$T_c = t_c * A \tag{21}$$

where A is the amount of transferring data (the number of bytes) and t_c is the time spent to transfer one byte of data. Note that here t_c is assumed constant over time. In practice, it is dependent on the network bandwidth and the communication latency. However, as pointed out in [6], for the majority of modern communication systems, the assumption of a fixed t_c can be used confidently. Note that in [6], the author considered the communication time between the master and one slave as fixed (or as an unit of time). However, in our model, the data sent from a node to master can be different from one to an other, so that we use the unit at the smaller scale.

By definition, the speedup is the ratio between the computation time for the system to run in serial mode and that in parallel mode [1]. Therefore, it is determined as the following:

$$S = \frac{GNt_f}{\frac{GNt_f}{P} + t_c * A} \tag{22}$$

Let $\alpha = \frac{t_f}{t_c}$, Eq. 22 can be rewritten as follows:

$$S = P\frac{GN\alpha}{GN\alpha + AP} \tag{23}$$

As shown above, A is a linear function of the number of slaves P and is defined explicitly depending on different architecture and communication schemes. In general, A can be written as follows:

$$A = a_1P + a_2 \tag{24}$$

Hence, S is rewritten as below:

$$S = P\frac{GN\alpha}{a_1P^2 + a_2P + GN\alpha} \tag{25}$$

From here, it is easy to obtain the optimal value for P by solving the equation $\frac{\partial S}{\partial P} = 0$:

$$P^* = \sqrt{\frac{GN\alpha}{a_1}} \tag{26}$$

The existence of the optimal value P^* indicates that the use of multiple nodes is not for free. The more nodes added, the more communication is imposed on the system; also, probably more idle time during the synchronization period. However, it also is clear that the optimal value is dependent on the value of α; as α increases, the more nodes can be added.

Here, an example is given using the master/slave architecture and CM2. For it, the speedup is rewritten as follows:

$$S = P\frac{GN\alpha}{\frac{16Gn}{f}P^2 + (A_0 - \frac{16Gn}{f})P + GN\alpha} \tag{27}$$

Therefore, $a_1 = \frac{16Gn}{f}$ and $a_2 = A_0 - \frac{16Gn}{f}$.

An example (where $P = 60$, $G = 100$, $N = 2000$, A=100 Kb, and α ranges from 1 to 1000) is used to calculate the dependency of the speedup to the number of slaves (also, $f = 1$ and $n = 2$). All the results are visualized in Figure 4. From the figure, once can clearly see that the curve of speed up approaches the linear one as α gets increased. Further, the optimal value for P also increases following the ranging of α.

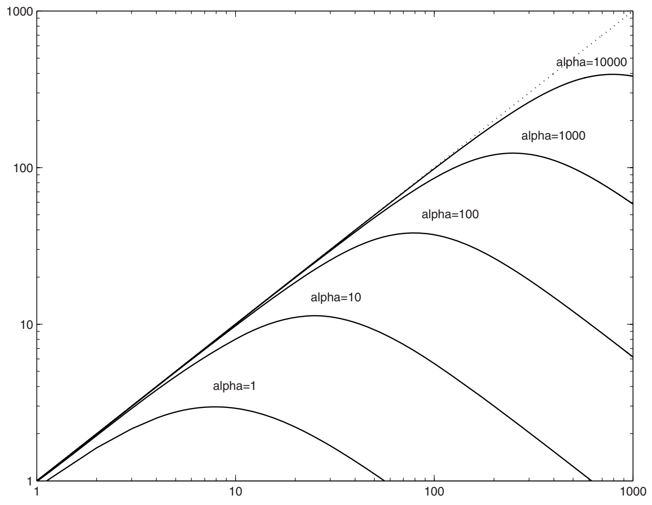


Fig. 4. An example of the effect of scaling on the speedup of the system given α= 1, 10, 100, 1000, and 10000

4 Experimental Studies

A series of experiments were conducted for analysis on the role of communication messages and explicit niching. Several problems were selected for the experiments, including DTLZ3 (2 objectives with multi-modality), DTLZ7 (2 objectives with discontinuity of the POF), ZDT4 (2 objectives with multimodality) and ZDT6 (2 objectives with non-uniformity of the POF). There are no particular reason for selecting these problems except they have been used widely in the literature. Due to the space limitation, the readers are referred to [9, 8] for more detailed descriptions of these problems. These problems were also used to investigate the performance of the distributed systems using local models with regard to problem scaling since they are scalable.

Here, each sub-population was assigned 100 individuals. The clustering architecture was tested with two configurations: 2 clusters (**2C**) and each was associated with 3 slaves(sub-populations); and 3 clusters (**3C**), with 2 sub-populations for each. The total number of sub-populations was 6. The other architectures (master/slave - **MS**, and island - **IS**) also were assigned 6 sub-populations (although later on we also tested with more than 6 sub-populations for the scalability of the system). Updating frequency was set as 5 generations.

Our experiments were carried out on a SGI-Altix system with a maximal number of 16 processors (Itanium-2 1.3 GHz). The total system shared-memory is 32Gb. Processors connect with each other and to the system memory via a network using NUMAlink-4 technology with a very high speed and a low latency (bandwidth of 6.4Gb/s and hardware latency of several hundreds of nanoseconds). We also used OpenMP technology to develop our distributed system.

4.1 Effect of Distributed Architectures and Communication Messages

In general, the performance of parallel systems is dependent on two factors: effectiveness and efficiency. For LOD-EMOS, the effectiveness is measured by the convergence rate and the quality of the obtained non-dominated sets. For this, the number of evaluations, required by approaches to reach a highly predefined level of the hypervolume ratio [8]- HR (0.999, for example), is used. The high level of HR is used to guarantee the quality of the non-dominated set and the recorded results also reveal how fast the approaches are in advancing to the global POF. Meanwhile, the efficiency is shown via the level of speedup that LOD-EMOS gained in comparison with its serial counterpart. The speedup will be calculated following Eq. 23.

Firstly, the number of evaluations that approaches spent to reach the predefined level of HR were reported in Table 1. These results will be used to analyze how the change of architecture affects the performance of LOD-EMOS.

Table 1. The average number of evaluations (and the standard error) that each architecture needed to reach the HR level of 0.999. All communication types were used. Symbol † indicates that the difference between the master/slave and the others in each category of communication messages is significant. The texts in bold-style show the best among different architectures.

Com msg	Architectures	DTLZ3	DTLZ7	ZDT4	ZDT6
CM1	MS	12400±2371	13560±940	42380±3786	8720±438
	3C	12540±2296	**11580±758**†	42860±3345	8320±562†
	2C	**12340±3042**	11780±431†	**40180±4464**	**8100±540**†
	IS	12880±2760	14780±1246†	41040±6146	8860±464
CM2	MS	12380±2843	14320±1187	42660±4173	8960±629
	3C	**12160±2441**	11420±912 †	41080±6290	8320±645†
	2C	12800±3227	**10740±824** †	41560±3509	**8220±391**†
	IS	12880±2760	14780±1246	**41040±6146**	8860±464
CM3	MS	12440±2337	14100±1144	42860±3878	8800±481
	3C	**11980±2152**	11300±864 †	41040±4514	8340±554†
	2C	12300±2260	**10880±799** †	40540±4374	**8240±567**†
	IS	12760±3099	14620±1304	**40540±3804**†	8680±492

It is clear that the clustering architecture was the best, while the island one showed the worst. This fact can be explained in the way that for clustering, the use of more than one interconnected global archive might give the local models more diverse information about the global trend than the single one. Meanwhile, the use of ring topology in the island architecture might slightly reduce the usefulness of the globally guided information together with PSO rules. That is why it required slightly more evaluations.

Regarding the effect of communication messages, it will be elaborated on all test architectures. Although the use of direction might lose some information, the overall performance is still the same as that of the method of using the non-dominated set (CM1). This result is confirmed for all architectures. As mentioned in the previous section, for CM1, the local model that contributes more solutions will be more dominating in the global information. This might not happen in the case of using direction (CM2 and CM3). However, for weighted direction (CM3), the effect of domination is somewhat regained. That is why it had better performance than that of using direction only; even it outperformed the method of using non-dominated set in some cases. However, it should be noted that the use of direction is expected to get no worse results than that of CM1, while helping to reduce the amount of transferred data.

In general, all approaches reached the global POF within 50 000 evaluations. Therefore, it becomes the checkpoint for investigating the speedup that LOD-EMOS obtained with each approach. Firstly, the amount of data (the number of Kbytes) transferred during the optimization process was measured and reported in Table 2. For CM1, it is obvious that the island architecture transferred the least amount of data as expected. For every generation, each

Table 2. The number of KBytes transferred (and the standard error). Symbol † indicates that the difference between the master/slave and the others is significant. The texts in bold-style show the best among different architectures.

Com msg	Architectures	DTLZ3	DTLZ7	ZDT4	ZDT6
CM1	MS	179.944±15.093	1114.483±15.813	134.831±13.038	659.972 ±6.972
	3C	631.538±20.499†	1617.720± 13.469†	556.225±10.271†	878.031 ±6.849†
	2C	659.581±22.853†	1840.866± 14.517†	591.016±12.974†	1011.644 ±8.810†
	IS	**8.766±0.000**†	**16.734 ±0.000**†	**7.969±0.000**†	**7.969±0.000**†
CM2	MS	17.503 ±0.529	37.237 ± 1.176	15.305 ±0.387	15.513 ±0.320
	3C	40.376 ±0.435†	79.499 ± 0.826†	36.354 ±0.272†	36.518 ±0.266†
	2C	42.528 ±0.420†	84.246 ± 1.131†	38.237 ±0.350†	38.331 ±0.305†
	IS	**8.766±0.000**†	**16.734 ±0.000**†	**7.969±0.000**†	**7.969±0.000**†
CM3	MS	17.659 ±0.529	37.393 ± 1.176	15.461 ±0.387	15.669 ±0.320
	3C	40.564 ±0.435†	79.687 ± 0.826†	36.542 ±0.272†	36.706 ±0.266†
	2C	42.715 ±0.420†	84.434 ± 1.131†	38.424 ±0.350†	38.518 ±0.305†
	IS	**8.965±0.000**†	**16.934 ±0.000**†	**8.168±0.000**†	**8.168±0.000**†

local model sent only an array of centroid's coordinates. Meanwhile, the master/slaves and clustering architectures sent more data since they work with the set of non-dominated solutions; further, the clustering one also includes the communication data among masters.

The story is much more different in the cases of using direction (CM2 and CM3). Sending data now is just about the direction of improvement - a record of several numerical values (the same as the record of a centroid). That is why the amount of data is significantly reduced (from 10 to 30 times). CM3 sent slightly more data than CM2 did. That is because it also sent the size of the non-dominated set for the weighting purpose (2-bytes data type). In general, the island architecture still sent the least amount of data.

Note that Eq. 23 formulates the speedup related to the amount of transferred data, the number of nodes and α - the ratio between the time require to evaluate an individual and the time to send one byte of data on the network. Clearly, this equation indicates the dependency of the speedup on the ratio α. There might be very little speedup (therefore, little efficiency) if the evaluation time is similar to the communication time. For this, α was tested with both small and large values (1 and 100). All estimated results were reported in Table 3 where all high speedups (that are equivalent with the efficiencies of more than 80%) are highlighted . By "estimated results" we mean Eq. 23 was used for this since the amount of data transmitted in each case was already available for calculation.

It is obvious that in the case of $\alpha = 1$, the speedup of all approaches was small (except island architectures since it works with the centroid rather than the whole set) and especially was less than 1.0 for CM1. However, this speedup increased as the ratio became higher (approaching the linear speedup that is 6.0 in this case). Among these, the clustering architecture seemed to have the least efficiency. While the island architecture had already the speedup of 3.0 when $\alpha = 1$, the clustering had less than 1.0 for all types

Table 3. The estimated speedup of LOD-EMOS (and the standard error) with different architectures and communication styles and using different α. Texts in bold-style show the high speedup (approaching linear level 6.0).

α	Com msg	Architectures	DTLZ3	DTLZ7	ZDT4	ZDT6
1	CM1	MS	0.260±0.021	0.043±0.001	0.343±0.030	0.073±0.001
		3C	0.076±0.002	0.030±0.000	0.086±0.002	0.055±0.000
		2C	0.073±0.002	0.026±0.000	0.081±0.002	0.048±0.000
		IS	2.883±0.000	1.958±0.000	3.026±0.000	3.026±0.000
	CM2	MS	1.900±0.039	1.073±0.028	2.078±0.034	2.059±0.028
		3C	1.003±0.009	0.555±0.005	1.094±0.007	1.090±0.007
		2C	0.961±0.008	0.527±0.006	1.049±0.008	1.047±0.007
		IS	2.883±0.000	1.958±0.000	3.026±0.000	3.026±0.000
	CM3	MS	1.888±0.039	1.070±0.027	2.064±0.034	2.046±0.027
		3C	0.999±0.009	0.554±0.005	1.089±0.007	1.085±0.006
		2C	0.957±0.008	0.526±0.006	1.045±0.008	1.043±0.007
		IS	2.849±0.000	1.942±0.000	2.988±0.000	2.988±0.000
100	CM1	MS	**4.911±0.075**	2.527±0.021	**5.145±0.070**	3.307±0.016
		3C	3.373±0.048	2.003±0.011	3.558±0.027	2.880±0.012
		2C	3.309±0.051	1.834±0.010	3.470±0.032	2.669±0.013
		IS	**5.936±0.000**	**5.879±0.000**	**5.942±0.000**	**5.942±0.000**
	CM2	MS	**5.873±0.004**	**5.736±0.008**	**5.889±0.003**	**5.887±0.002**
		3C	**5.715±0.003**	**5.464±0.005**	**5.742±0.002**	**5.741±0.002**
		2C	**5.701±0.003**	**5.435±0.007**	**5.730±0.002**	**5.729±0.002**
		IS	**5.936±0.000**	**5.879±0.000**	**5.942±0.000**	**5.942±0.000**
	CM3	MS	**5.872±0.004**	**5.735±0.008**	**5.888±0.003**	**5.886±0.002**
		3C	**5.714±0.003**	**5.463±0.005**	**5.741±0.002**	**5.740±0.002**
		2C	**5.700±0.003**	**5.434±0.007**	**5.728±0.002**	**5.728±0.002**
		IS	**5.934±0.000**	**5.877±0.000**	**5.940±0.000**	**5.940±0.000**

of communication. It is interesting to see that the speedup in the cases of multi-modal problems (DTLZ4 and ZDT4) was much more than that of the unimodal problems (DTLZ7 and ZDT6) for CM1. That is because, for unimodal problems, the local models easily found the POF and the size of the non-dominated sets increased quickly. Therefore, the amount of data also increased consequently and the system spent more time on communication.

In general, the use of direction (CM1 and CM2) gives LOD-EMOS much more efficiency in comparison to their counterpart - CM1. It is important to note that for practical problems (also using the modern networked systems that can offer the transmission speed of Gigabits), the ratio α is much higher than the ones reported in this analysis. In our share-memory system, the ratio can be up to thousands. More detailed analysis will be given in Section 4.3.

4.2 Effect of Frequencies

In this section, another factor in the design of a distributed system is analyzed: *frequency*. Three levels of frequency were tested including: 1, 5 and 10 generations. The use of *one-generation* frequency is to create an upper bound for communication. For this case, the nodes communicate in every generation. From the communication perspective, this will cause heavier workload, especially for CM1 where all non-dominated solutions are sent during the optimization process. Further, for the system performance, it might be not good for solving multi-modal problems since as the effect of global information is too frequent, not much time is allowed for nodes to discover their own niches. Therefore, the global information might quickly get poor.

Table 4. The average number of evaluations (and the standard error) that each architecture needed to reach the HR level of 0.999. Frequency is **one generation**. Symbol † indicates that the difference between the master/slave architecture and the others is significant. The texts in bold-style show the best approaches among different architectures.

Com msg	Architectures	DTLZ3	DTLZ7	ZDT4	ZDT6
CM1	MS	**12240±2499**	11080 ±1133	43120±4334	6500±612
	3C	13320±2615†	**8180 ±826** †	**40620±3031**†	**5760±642**†
	2C	**13040±2832**	**8180 ±696** †	41080±4825	5780±485†
	IS	**12700±3719**	10300 ±1081†	49680±5354†	7500±859†
CM2	MS	**12740±2705**	9820 ±826	**43300±3847**	6560±609
	3C	**13780±3454**	5820 ±790 †	54600±6406†	5820±758†
	2C	14060±3243†	**4800 ±315** †	56960±7977†	**5380±841**†
	IS	**12700±3719**	10300 ±1081	49680±5354†	7500±859†
CM3	MS	**12960±2709**	10960 ±1382	**47420±5654**	6820±1120
	3C	**14300±4073**	6040 ±945 †	55460±5654†	5920±955†
	2C	**13580±3855**	**4940 ±464** †	**56920±8775**	**5440±833**†
	IS	**13660±3304**	9700 ±850 †	48000±5584†	7320±1002

To investigate the trade-off effect of changing frequency to the system's effectiveness, again, the number of evaluations that approaches needed to reach the predefined level of HR is employed and they all were reported in Tables 1, 4, 5. From these results, the tradeoff is quite clear. For unimodal problems (DTLZ7 and ZDT6), the system got worse as the frequency increased. There are no traps in these problems, the more interaction among local models, the more benefit the slow ones gained from the faster counterparts. Overall, the system got faster convergence. For multi-modal ones (DTLZ3 and ZDT4), the effect is opposite. The case of a frequency of one generation had the worst performance where the number of evaluations was much more than that of the other cases, especially for problem ZDT4. However, the benefit of reducing frequency is not infinite. The system obtained the best performance when using frequency of 5 generations; when it changed to 10 generations, the performance of the system downgraded. This finding is also consistent with the majority of research results in the literature [6] where the frequency is usually set at a relatively small value.

For the matter of transferring data, it is obvious from Tables 2, 6, and 7 that when the frequency value got smaller (higher frequency), the amount of

Table 5. The average number of evaluations (and the standard error) that each architecture needed to reach the HR level of 0.999. Frequency is **ten generations**. Symbol † indicates that the difference between the master/slave architecture and the others is significant. The texts in bold-style show the best approaches among different architectures.

Com msg	Architectures	DTLZ3	DTLZ7	ZDT4	ZDT6
CM1	MS	**12640±2394**	16740±1172	**41840±3765**	**10040±686**
	3C	**12500±2359**	15860±1099†	**42200±4258**	**9860 ±490**
	2C	**12140±2341**	**15760±1113**†	**41940±3931**	**9880 ±584**
	IS	**12780±2623**	17100±1896	**42700±4674**	**10000±530**
CM2	MS	**12740±2750**	17480±1248	**41880±3528**	**10120 ±645**
	3C	**12420±2707**	16140±1037†	**43580±3536**	**9860 ±604**
	2C	**12580±2469**	**15660±1244**†	**40940±4907**	**9860 ±681**
	IS	**12780±2623**	17100±1896	**42700±4674**	**10000 ±530**
CM3	MS	**13120±2896**	17660±1565	**41540±4522**	10280 ±645
	3C	**12400±2776**	16000±809 †	**41220±3686**	**9900 ±845**†
	2C	**12460±2238**	**15640±996** †	**40940±3308**	9920 ±625†
	IS	**12720±2355**	17500±1462	**41480±5054**	10060 ±663

Table 6. The total amount of data transferred during the optimization process (and the standard error). All communication types were used. Frequency is **one generations**. Symbol † indicates that the difference between the master/slave architecture and the others is significant. The texts in bold-style show the best approaches among different architectures.

Com msg	Architectures	DTLZ3	DTLZ7	ZDT4	ZDT6
CM1	MS	843.814 ±99.975	5708.303±132.825	624.750 ±57.624	3493.109±39.590
	3C	3034.287±69.508†	8078.058±82.631†	2732.219 ±53.651†	4467.084±22.295†
	2C	3206.561±77.906†	9282.782±133.506†	2863.231 ±65.738†	5182.572±21.640†
	IS	**42.797 ±0.000**†	**81.7030.0±00**†	**38.906 ±0.000**†	**38.906 0±.000**†
CM2	MS	74.221 ±0.529	145.518±1.176	66.867 ±0.387	67.076 0±.320
	3C	69.228 ±0.344†	134.679±1.030†	82.760 ±0.272†	82.924 0±.266†
	2C	99.246 ±0.420†	192.527±1.131†	89.799 ±0.350†	89.893 0±.305†
	IS	**42.797 ±0.000**†	**81.7030±.000**†	**38.906 ±0.000**†	**38.906 0±.000**†
CM3	MS	75.022 ±0.529	146.319±1.176	67.668 ±0.387	67.876 0±.320
	3C	92.384 ±0.435†	177.913±0.826†	83.721 ±0.272†	83.565 0±.266†
	2C	100.207 ±0.420†	193.488±1.131†	90.760 ±0.350†	90.854 0±.305†
	IS	**43.770 ±0.000**†	**82.6760±.000**†	**39.879 ±0.000**†	**39.879 0±.000**†

data transferred also increased. That is because if the system communicates more frequently, the more data will be sent over the network. Therefore, the workload will be increased.

In general, the implication of this investigation is about the relationship between frequency and the performance of the distributed system. For a frequent communication (such as 1 generation), the consequence is obvious. However, a high value of frequency (such as 10 in this study), might help to reduce the workload, but might also affect the quality of obtained solutions, since it causes the lack of sharing of useful information during the optimization process.

Table 7. The total amount of data transferred during the optimization process (and the standard error). Frequency is **ten generations**. Symbol † indicates that the difference between the master/slave architecture and the others is significant. The texts in bold-style show the best approaches among different architectures.

Com msg	Architectures	DTLZ3	DTLZ7	ZDT4	ZDT6
CM1	MS	96.289 ±9.809	555.460±12.331	75.384 ±9.605	332.738±4.440
	3C	331.454 ±6.867†	826.260±9.520†	298.741±6.467†	452.469±4.291†
	2C	353.163 ±10.315†	939.311±9.269†	314.638±7.786†	521.213±2.746†
	IS	**4.641 ±0.000†**	**8.859 ±0.000†**	**4.219 ±0.000†**	**4.219 ±0.000†**
CM2	MS	10.628 ±0.529	24.112 ±1.176	9.055 ±0.387	9.263 ±0.320
	3C	34.189 ±0.435†	67.687 ±0.826†	30.729 ±0.272†	30.893 ±0.266†
	2C	35.653 ±0.420†	71.121 ±1.131†	31.987 ±0.350†	32.081 ±0.305†
	IS	**4.641 ±0.000†**	**8.859 ±0.000†**	**4.219 ±0.000†**	**4.219 ±0.000†**
CM3	MS	10.706 ±0.529	24.190 ±1.176	9.133 ±0.387	9.341 ±0.320
	3C	34.283 ±0.435†	67.780 ±0.826†	30.823 ±0.272†	30.987 ±0.266†
	2C	35.746 ±0.420†	71.215 ±1.131†	32.081 ±0.350†	32.174 ±0.305†
	IS	**4.746 ±0.000†**	**8.965 ±0.000†**	**4.324 ±0.000†**	**4.324 ±0.000†**

4.3 Scalability and Speedup

This section is dedicated to an investigation of the scalability of LOD-EMOS. In Section IV, it has been shown that the scalability of LOD-EMOS depends very much on the ratio α; in other words, the optimal number of nodes is related to the value of α. Further, the proposed theoretical paradigm acts as a boundary for LOD-EMOS. The analysis given in this section serves as an experimental confirmation of this boundary.

The master/slave architecture was selected for the experiments on the problem of ZDT4. Again, the amount of data transferred during the optimization process (up to 50 000 evaluations) was recorded. The number of nodes ranged from 2 to 10. These results were reported in Table 8. Obviously, the approaches of using directions showed the least amount of data in all cases of the number of nodes as expected. It is ten times less than that of CM1. Further, the system got more data as the nodes increased for all types of communication messages. To get more understanding of how this affects the speedup, several values of α were tested and all results were visualized in Figures 5 and 6.

From Figure 5 (for CM1), it is easy to see that the speedup of the distributed system increased in all cases of nodes as α increased. While the speedup is less than one when $\alpha = 1$, it increased to linear speedup for a small number of nodes (and near-linear when there are about 9 or 10 nodes) when α reached 1000. As previously pointed out, CM1 usually sends a quite large amount of data (the whole non-dominated set) among nodes. This might affect the speedup of the system.

Further, the scalability of the system is also related to the degree of α. It is clear from the figure that as α was small, the optimal number of nodes is

Table 8. The amount of data transferred (and the standard error) with different cases of the number of nodes. Symbol † indicates that the difference between 2 nodes and others is significant. The texts in bold-style show the best case of the number of nodes.

Nodes	CM1	CM2	CM3
2	141.131±27.797	**3.201 ±0.185**	**3.232 ±0.185**
3	149.116±32.412	6.201 ±0.219†	6.263 ±0.219†
4	145.159±25.539	9.286 ±0.265†	9.380 ±0.265†
5	136.778±19.990	12.247 ±0.358†	12.372 ±0.358†
6	**134.831±13.038**	15.305 ±0.387†	15.461 ±0.387†
7	150.503±15.141	18.174 ±0.319†	18.362 ±0.319†
8	163.784±12.717†	21.070 ±0.352†	21.289 ±0.352†
9	170.959±12.274†	24.094 ±0.442†	24.344 ±0.442†
10	176.638±14.108†	27.005 ±0.529†	27.286 ±0.529†

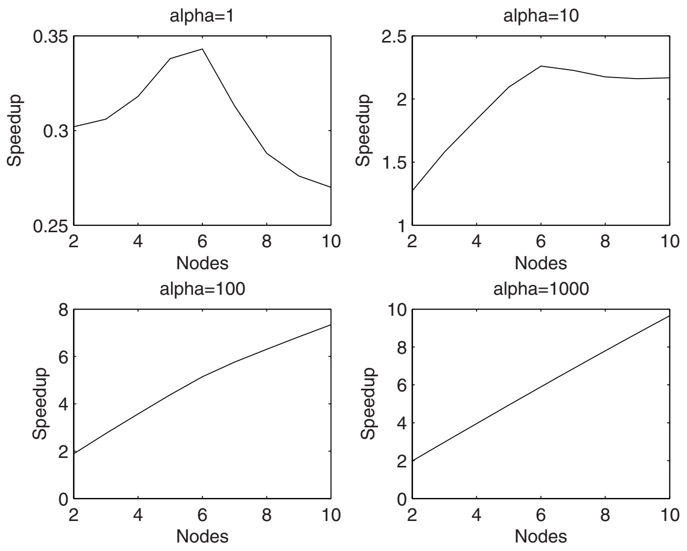


Fig. 5. The estimated speedup for Master/slave architecture using CM1

about 6. As α increased, the optimal number of nodes also increased. This is consistent with the theoretical finding in the previous section. This again indicates that there is is a very little advantage of using distributed system for problems with cheap objective evaluation.

For CM2 and CM3 (An example is given for CM2 in Fig. 6), the speedup was much more than that of CM1 regardless of the number of nodes; for example, when $\alpha = 1$, their speedup was more than one in all cases, while for CM1, it was less than one. Further, they certainly reached the linear speedup when $\alpha = 1000$. Although their peak was smaller (4 nodes) when $\alpha = 1$, this value increased quickly for more than 10 as $\alpha = 10$.

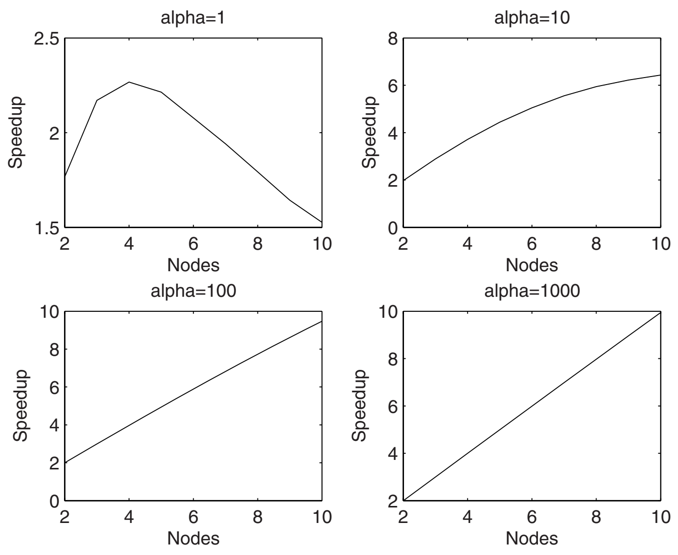


Fig. 6. The estimated speedup for Master/slave architecture using CM2

We now focus on the time of LOD-EMOS on our actual SGI-Altix multiprocessor system. In this system, sending solutions is just the matter of writing/reading memory addresses. Therefore, the time is very fast and the communication-related time is dominated by the factors of creating/schyncronizing threads of OpenMP that might be not much different between communication styles. We measured the wall-time from the start to the end of the optimization process, therefore, it might include some extra time from the operating system (such as scheduling time). Also, the evaluation for ZDT4 is also very fast (about 1.8 microseconds for the evaluation of an individual). To make more sense, we increased the evaluation time of an individual to nearly 0.1 millisecond (about 0.07-0.08 millisecond) by adding a simple loop. Again, the results were averaged from 30 runs with different random seeds. For each run, 10 trials were performed to get the least time (in order to minimize the effect of the operating system).

The results were visualized in Figure 7 as the time versus the number of nodes. For the figure, we can see that as the number of nodes increased, the time decreased. Among communication styles, the time of CM2 and CM3 is obviously less than that of CM1. In slower systems, this difference is certainly higher since the sending of solutions for CM1 will cause more time than CM2 and CM3 (that require sending only the statistical information). Further, the speed of reduction became smaller (especially CM2 and CM3) when the number of nodes increased to 10. That is because when the number of nodes increased, more the communication time was also added to the total time.

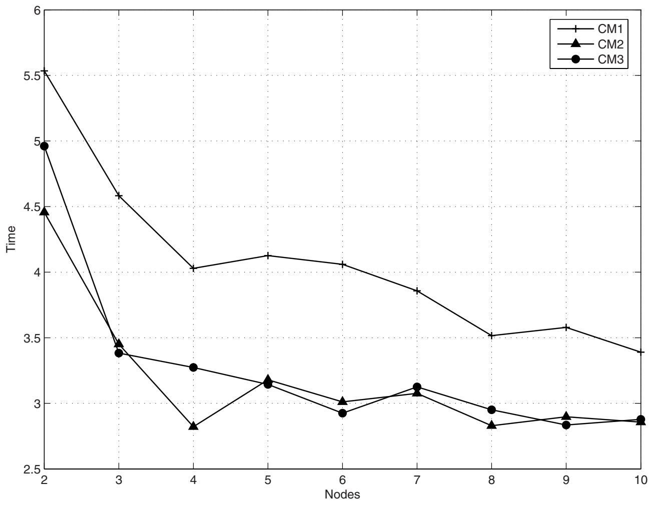


Fig. 7. Computation time of LOD-EMO with OpenMP on SGI-Altix system

5 Conclusion

In this chapter, the framework of local models was investigated under the effects of different aspects, including architectures, communication and scalability. A system called LOD-EMOS was built from the framework. Three architectures were proposed for LOD-EMOS: master/slave, island and clustering. Three different types of communication messages were applied to LOD-EMOS. Further, it was also tested under different schemes for frequency.

From the investigation, it has been found that: (1) Local models can be adapted easily to different distributed architectures; (2) Direction of improvement can be used to determine the global centroid and therefore can be used as communication messages; (3) Localization offers a good way to obtain the direction of improvement; (4) Local models have a capability of dealing with scalability as the number of nodes increases (an optimal value was been pointed out).

The developed theory in this chapter is devoted to understanding the role of communication messages and explicit niching in distributed MOEAs. LOD-EMOS was used as the platform for the investigation. The built paradigms (the equations) are approximate and easy to be adapted to particular situation since they show the boundary for the practical systems. Further, as pointed out in [6] it is usually difficult to build the exact paradigms, and if they are built, they are expensive to use and hence have little practical significance.

References

1. Akl, S.G.: The Design and Analysis of Parallel Algorithms. Prentice-Hall, Englewood Cliffs (1991)
2. Banos, R., Gil, C., Paechter, B., Ortega, J.: Parallelization of population-based multi-objective meta-heuristics: An empirical study. Applied Mathematical Modelling 30, 578–592 (2006)
3. Branke, J., Schmeck, H., Deb, K., Maheshwar, R.S.: Parallelizing multiobjective evolutionary algorithms: Cone separation. In: Congress on Evol. Comp., pp. 1952–1957. IEEE Press, Los Alamitos (2004)
4. Bui, L.T., Abbass, H.A., Essam, D.: Local models: An approach to disibuted multi-objective optimization. Computational Optimization and Applications 42(1), 105–139 (2009)
5. Bui, L.T., Deb, K., Abbass, H.A., Essam, D.: Interleaving guidance in evolutionary multi-objective optimization. Journal of Computer Science and Technology 23(1), 44–66 (2008)
6. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer, Boston (2000)
7. de Toro, F., Ortega, J., Ros, E., Mota, S., Paechter, B., Martin, J.M.: Psfga: Parallel processing and evol. comp. for multiobjective optimisation. Parallel Computing 30, 721–739 (2004)
8. Deb, K.: Multiobjective Optimization using Evolutionary Algorithms. John Wiley and Son Ltd., New York (2001)
9. Deb, K., Thiele, L., Laumanns, M., Zitzler, E.: Scalable test problems for evolutionary multi-objective optimization, TIK-Report no. 112. Technical report, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), Zurich (2001)
10. Deb, K., Zope, P., Jain, A.: Distributed computing of pareto optimal solutions with evolutionary algorithms. In: Fonseca, C.M., Fleming, P.J., Zitzler, E., Deb, K., Thiele, L. (eds.) EMO 2003. LNCS, vol. 2632, pp. 534–549. Springer, Heidelberg (2003)
11. Eberhart, R.C., Shi, Y.: Particle swarm optimization: developments, applications and resources. In: Congress on Evol. Comp. IEEE Press, Los Alamitos (2001)
12. Essabri, A., Gzara, M., Loukil, T.: Parallel multi-objective evolutionary algorithm with multi-front equitable distribution. In: Fifth Int. Conf. on Grid and Cooperative Computing (GCC 2006), pp. 241–244. IEEE Computer Society, Los Alamitos (2006)
13. Flores, S.D., Cegla, B.B., Cceres, D.B.: Telecommunication network design with parallel multi-objective evolutionary algorithms. In: 2003 Conf. on Towards a Latin American agenda for network research, pp. 1–11. ACM Press, New York (2003)
14. Hiroyasu, T., Miki, M., Wantanabe, S.: The new model of parallel genetic algorithm in multiobjective optimization problems- divided range multi-objective genetic algorithm. In: Congress on Evol. Comp., pp. 333–340. IEEE Press, Los Alamitos (2000)
15. Jaimes, A.L., Coello, C.A.C.: MRMOGA: Parallel evolutionary multiobjective optimization using multiple resolutions. In: Congress on Evol. Comp., pp. 2294–2301. IEEE Press, Los Alamitos (2005)

16. Jones, B.R., Crossley, W.A., Lyrintzis, A.S.: Aerodynamic and aeroacoustic optimization of airfoils via a parallel genetic algorithm. In: 7th Symposium on Multidisciplinary Analysis and Optimization, pp. 1088–1096. AIAA (1998)
17. Makinen, R., Neittaanmaki, P., Periaux, J., Sefrioui, M., Toivanen, J.: Parallel genetic solution for multiobjective mdo. In: Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers, pp. 352–359. Elsevier, Amsterdam (1997)
18. Marco, N., Lanteri, S., Desideri, J.-A., Périaux, J.: A parallel genetic algorithm for multi-objective optimization in computational fluid dynamics. In: Evolutionary Algorithms in Engineering and Computer Science, pp. 445–456. John Wiley & Sons, Ltd., Chichester (1999)
19. Mehnen, J., Michelitsch, T., Schmitt, K., Kohlen, T.: pMOHypEA: Parallel evolutionary multiobjective optimization using hypergraphs. Technical Report CI-187/05, the Collaborative Research Center, University of Dortmund (2005)
20. Obayashi, S., Sasaki, D., Takeguchi, Y., Hirose, N.: Multiobjetive evol. comp. for supersonic wing-shape optimization. IEEE Trans. on Evol. Comp. 4(2), 182–187 (2000)
21. Quagliarella, D., Vicini, A.: Sub-population policies for a parallel multiobjective genetic algorithm with applications to wing design. In: IEEE Int. Conf. On Systems, Man, And Cybernetics, pp. 3142–3147. IEEE Press, Los Alamitos (1998)
22. Rowe, J., Vinsen, K., Marvin, N.: Parallel gas for multiobjective functions. In: Second Nordic Workshop on Genetic Algorithms and Their Applications, pp. 61–70. University of Vaasa, Finland (1996)
23. Sait, S.M., Faheemuddin, M., Minhas, M.R., Sanaullah, S.: Multiobjective vlsi cell placement using distributed genetic algorithm. In: 2005 Conf. on Genetic and Evol. Comp., pp. 1585–1586. ACM Press, New York (2005)
24. Stanley, T.J., Mudge, T.: A parallel genetic algorithm for multiobjective microprocessor design. In: The Sixth Int. Conf. On Genetic Algorithms, pp. 597–604. Morgan Kaufmann Publishers, San Francisco (1995)
25. Streichert, F., Ulmer, H., Zell, A.: Parallelization of multiobjective evolutionary algorithms using clustering algorithms. In: Coello Coello, C.A., Hernández Aguirre, A., Zitzler, E. (eds.) EMO 2005. LNCS, vol. 3410, pp. 92–107. Springer, Heidelberg (2005)
26. Talbi, E.-G., Meunier, H.: Hierarchical parallel approach for gsm mobile network design. Journal of Parallel Distributed Computing 66(2), 274–290 (2006)
27. Tan, K.C., Yang, Y.J., Goh, C.K.: A distributed cooperative coevolutionary algorithm for multi-objective optimization. IEEE Trans. on Evol. Comp. 10(5), 527–549 (2006)
28. Tan, K.C., Yang, Y.J., Lee, T.H.: Designing a distributed cooperative coevolutionary algorithm for multiobjective optimization. In: Congress on Evol. Comp., pp. 2513–2520. IEEE Press, Los Alamitos (2003)
29. Veldhuizen, D.A.V., Zydallis, J.B., Lamont, G.B.: Considerations in engineering parallel multiobjective evolutionary algorithms. IEEE Trans. on Evol. Comp. 7(2), 144–173 (2003)
30. Xiao, N., Armstrong, M.P.: A specialized island model and its application in multiobjective optimization. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Kendall, G., Wilson, S.W., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A., Dowsland, K.A., Jonoska, N., Miller, J., Standish, R.K. (eds.) GECCO 2003. LNCS, vol. 2724, pp. 1530–1540. Springer, Heidelberg (2003)

31. Xiong, S., Li, F.: Parallel strength pareto multi-objective evolutionary algorithm for optimization problems. In: IEEE Congress on Evol. Comp., pp. 2712–2718. IEEE Press, Los Alamitos (2003)
32. Xu, K., Louis, S.J., Mancini, R.C.: A scalable parallel genetic algorithm for x-ray spectroscopic analysis. In: 2005 Conf. on Genetic and Evol. Comp., pp. 811–816. ACM Press, New York (2005)
33. Zhu, Z.-Y., Leung, K.-S.: Asynchronous self-adjustable island genetic algorithm for multi-objective optimization problems. In: Congress on Evol. Comp., pp. 837–842. IEEE Press, Los Alamitos (2002)

Adaptive Scheduling Algorithms for the Dynamic Distribution and Parallel Execution of Spatial Agent-Based Models

Matthias Scheutz and Jack Harris

Abstract. In previous work [7], we proposed a general framework for defining agent-based models (ABMs) and introduced two algorithms for the automatic parallelization of agent-based models: a general version **P-ABM**$_G$ for all ABMs definable in the framework and a more specific variant **P-ABM**$_S$ for "spatial ABMs", which can utilize the additional spatial information to obtain performance improvements. Both algorithms can automatically distribute ABMs over multiple processors and dynamically adjust the degree of parallelization based on available computational resources throughout the simulation runs. However, they are not sensitive to inefficiencies in the sequence in which agents in each parallel simulation instance are updated.

In this chapter, we introduce a minimal framework for describing ABMs and propose various asynchronous scheduling algorithms for agent-based simulations that address the update ineffficencies of simulation schedulers. The proposed algorithms work in conjunction with **P-ABM**$_G$ and **P-ABM**$_S$ and allow for efficient simulation runs that can automatically and better utilize the asynchronous nature of parallel distributed agent-based simulations (including split-ups of specific simulation models and dynamic load-balancing). We demonstrate the significant performance gains of the proposed algorithms using an actual agent-based model used for studying female choice and foraging in biological research.

1 Introduction

Simulations of agent-based models (ABMs) have been successfully applied in a variety fields to reveal and elucidate interaction patterns among entities in complex

Matthias Scheutz · Jack Harris
Human-Robot Interaction Laboratory
Cognitive Science Program and School of Informatics
Indiana University
Bloomington, IN 47406
e-mail: {mscheutz,jackharr}@indiana.edu

systems that are otherwise difficult to detect and understand. Depending on the investigated problem, the entities – or "agents"[1] – in agent-based models might take a different form. Model varieties range from chemicals or simple cybernetic creatures in artificial life, to web pages, computers, or human users in complex networks, to game-theoretic players in economics, to groups of humans or animals in social studies, biology, and anthropology (see [7] for more references).

What is common to all these diverse models is that they decompose the behavior of complex systems into tractable actions and interactions of individual agents. This is typically achieved by defining *rules* that determine the behavior of individual agents for all possible contexts in which the agent might find itself. A special class of agent-based models, the *spatial* agent-based models, explicitly defines an *environment*, which is typically a metric space. Every agent is situated in a particular location in the environment at any given time, but the location may change over time (e.g., if the agent is moving). Every agent also has an *interaction range* that, given the agent's location, determines the set of other agents in the environment with which it can interact at any given time; it cannot interact with or have any effects on agents outside of its interaction range.[2]

From a computational perspective, agent-based models are interesting because they often lend themselves to efficient, parallel implementations. One obvious way to parallelize agent-based models, for example, is to spawn a separate computational thread for each agent in a given simulation. These computational processes will take care of computing the agent's behavior and will of course have to be synchronized to ensure that all agents are updated consistently – we will say more about this shortly. Since for typical models, the number of agents in the model will by far outnumber the cores or processors available on the computer running the model simulations, it might also make sense to distribute simulations across multiple computers to better utilize the intrinsic parallelism in agent-based models. There is, however, a critical difference between distributing simulations over multiple connected computers and parallelizing a given model within one computational process on one computer (e.g., using a parallel programming language like ADA, or a threaded programming language like JAVA). In single process simulations, all agents can access the same environment and can use synchronization mechanisms available within the process, while in distributed simulations the environment has to be replicated on each host computer, and synchronization between agents and environments have to be achieved using networked "inter-process" synchronization primitives. Whereas parallelism within one computational process might be already implemented in the employed simulation environment, dynamic parallelization via distribution of the

[1] Agent-based models–sometimes also called "individual-based" models–are often used to simulate the behavior of complex real-world systems. They are used when possible state changes of individual entities are known and can be encoded in rules, while no such knowledge exists for global world states (e.g., the state given by the environment and all its agents).

[2] Note that general agent-based models can be viewed as a special case of spatial agent-based models in that all agents are located in the same location and can interact with all other agents at any given time.

environment over multiple computers is not available in any of the common modeling environments. Augmenting such environments to support multi-host distribution requires significant programming expertise, which modelers usually do not possess or are unwilling to invest their time, given that their research interest lies in the simulation results and not the computational infrastructure.[3] We believe that modelers should not have to worry about computational issues, but should be able to define their models in their favorite modeling environment, and the computational infrastructure should take care of running these models in the most efficient fashion given the available resources.

We have previously developed algorithms that will support modelers in achieving good turn-around times of model simulations by automatically parallelizing and distributing general and spatial agent-based models [7]. In this chapter, we extend our previous ideas for scheduling agents in simulations of spatial agent-based models by introducing novel scheduling algorithms. These algorithms take advantage of both the inherent parallelism in agent-based models and the interaction ranges of agents in spatial models. We demonstrate that these algorithms can achieve a significant performance improvement over standard scheduling algorithms in the context of our previous parallel and distributed algorithms [7] through a reference implementation in our SWAGES system [8]. This improvement is achieved by virtue of tightly integrating the simulation scheduler with the distribution algorithm. While the proposed algorithms will already be of great utility for modelers, they also pose a variety of interesting open problems for future research, which we will briefly address at the end of our exposition.

2 Distributed Simulations of Agent-Based Models

Various kinds of formalisms and frameworks have been developed to capture this diversity of agent-based models (e.g., some models are essentially physics-based, while others operate solely on a social level). We have previously attempted to define general formal frameworks for hierarchical [6] and spatial [7] agent-based models that were intended to be maximally inclusive. Here, we take the opposite approach and attempt to make due with the smallest formal framework for spatial agent-based models that is sufficient for defining and employing our distribution and scheduling algorithms.

We start with the assumption that each spatial agent-based model (S-ABM) $\mathscr{M}$ has an environment $\langle Env_{\mathscr{M}} \rangle$ that can be modeled as a discrete or continuous *metric* space (e.g., with the Euclidean norm). Such models allow for the simulation of interactions among agents based on a notion of *distance*. Spatial distance is not only crucial for understanding the behavior of many biological systems and organizations of agents in physical spaces (e.g., insect swarms, flocks of birds, schools of

[3] While we don't have formal evidence for this claim, in our experience modelers are happy to put up with very long single computer simulation runs, before they are willing to entertain the possibility of having to manually distribute their models.

fish, etc.), but it is also essential for the parallelization and distribution algorithms we will review in this chapter. We also assume that for each agent A in $\mathscr{M}$ there is a clearly defined *interaction range* I_A that, given the position of all agents in the environment, determines at any given time the set of agents it can (potentially) interact with at that time.[4] Finally, we assume that each model has a set of initial conditions $Init_{\mathscr{M}}$ and a set of terminating conditions $Term_{\mathscr{M}}$ which determine the initial and final states of a simulation of model $\mathscr{M}$ under those conditions. A (discrete-event) simulation of $\mathscr{M}$ is a sequence of updates of all agents A such that for any point in the sequence and any agent, all agents within its interaction range have had the same number of updates since the start of the simulation. The rationale for this definition will become clear later, for now just note that the standard updating sequence in discrete-time simulations falls under this defintion. This sequencing which we call "cycle-based update strategy" updates every agent in A before starting over and updating every agent in A again, and repeatedly looping through the set of agents until a terminating condition is reached.

2.1 A Minimal Framework of Agent-Based Models

We start by defining the notion of spatial agent-based model.

Definition 1 (Spatial agent-based model). *A* spatial agent-based model $\mathscr{M} = \langle Env_{\mathscr{M}}, ATypes_{\mathscr{M}}, Init_{\mathscr{M}}, Term_{\mathscr{M}} \rangle$ *consists of an n-dimensional bounded or unbounded metric space* $Env_{\mathscr{M}}$ *(consisting of locations that can be occupied by agents), a set of agent types* $ATypes_{\mathscr{M}}$*, a set of initial conditions* $Init_{\mathscr{M}}$*, and a set of terminating conditions* $Term_{\mathscr{M}}$*.* $Init_{\mathscr{M}}$ *is a set of agents and* $Term_{\mathscr{M}}$ *is a set of functions from the powerset of* $Agents_{\mathscr{M}}$ *(the set of all possible agents in* $\mathscr{M}$ *into* $\{true, false\}$*. An* agent A *is a triple* $\langle ID_A, Type_A, State_A \rangle$ *which consists of the agent's unique identifier* $ID_A \in \mathbb{N}$ *(required to be able to dissociate agents that would otherwise be identical with respect to their remaining information), an agent type* $Type_A \in ATypes_{\mathscr{M}}$*, and an agent state* $State_A$*, where* $State_A = \langle L_A, I_A, T_A, U_A, \ldots \rangle$ *contains the agent's location* $L_A \in Env_{\mathscr{M}}$ *in the environment, its interaction range* $I_A \in Env_{\mathscr{M}}$*, a translation function* T_A *which determines for a given location the maximum distance an agent can travel within one update,*[5] *an agent update function* U_A *mapping sets of agent states onto agent states, and any other pertinent information about the agent particular to the model.*[6]

[4] Note that the interaction range is allowed to change over time, but at any given point in time it has to be defined and uniquely determinable. Furthermore, note that we are not distinguishing between "sensory" and "actuator" ranges here, see [7] for such a distinction.

[5] We will discuss the reason for this function later.

[6] Note that formally the definitions of "agent-based model", "agent", and "agent state" are co-recursive, i.e., mutually dependent and thus mutually defined. This type of definition requires non-well-founded set theory as a formal framework, where the "Solution Lemma" ensures that these kinds of structures are properly defined and exist [1].

Equipped with the notion of agent, agent state, and agent-based model we can now define what we mean by a simulation of an agent-based model:

Definition 2 (Simulation of an S-ABM). *A* simulation $S_{\mathscr{M},C_0}$ *of an S-ABM* $\mathscr{M}$ *is defined as a finite sequence of configurations* $S_{\mathscr{M},C_0} = \langle C_0, C_1, \ldots, C_k \rangle$ *starting with configuration* C_0 *and ending with* C_k*. Each configuration* C_i $(0 \le i \le k)$ *is a set of agents of some type in* $ATypes_{\mathscr{M}}$ *– we will use* $Agents_{\mathscr{M}}$ *to denote the set of all possible agents supported by* $\mathscr{M}$*.* C_0 *is an initial configuration in* $Init_{\mathscr{M}}$*.* C_k *is a terminal condition such that for some function* $f \in Term_{\mathscr{M}}$ $f(C_k) = true$ *and there is no configuration* C_j *with* $j < k$ *and* $f \in Term_{\mathscr{M}}$ *such that* $f(C_j) = true$*. We also require for all configurations* C_i *and* C_{i+1} $(0 \le i < k)$ *to be "consistent", i.e., if* C_j *"follows"* C_i *in the simulation sequence (i.e.,* $j = i + 1$*) then* C_j *is obtained from* C_i *by (simultaneously) updating a subset of agents* $\mathscr{A} \subseteq C_i$ *such that all agents* $A' \in C_i$ *whose interaction range intersect with that of some* $A \in \mathscr{A}$ *are (already) in* $\mathscr{A}$*. An agent* $A \in C_i$ *(part of a simulation* $\langle C_0, C_1, \ldots, C_k \rangle$ *of S-ABM* $\mathscr{M}$*) is said to* be at the n-th cycle *if A has been updated n times, i.e.,* $U_A^n(State_A^0) = State_A$ *where* $State_A^0$ *was the state of A in* C_0 *and* $State_A^n$ *is its state after n updates. Finally, a configuration* C_i *is said to be* at cycle n *if all of its agents are at the n-th cycle.*

Note that simulations of agent-based models are, by definition, *consistent sequences* of configurations where all subsets of agents with intersecting interaction ranges are at the same cycle (this is a more permissive notion of configuration than the one implicitly underlying typical event-based simulations, namely that all agents in a configuration must be at the same cycle). Consequently, sequences of configurations that are *not consistent* (i.e., where agents get updated in an "inconsistent fashion" as would be the case if an agent that was updated twice already got updated based on its interaction with an agent that had only been updated once) are not simulations of agent-based models. However, the above definition of "consistent configuration" does not require that there be a "unique successor" of a given configuration (as is typically defined for discrete-event-based simulations), because for any given set of agents there could be many possible configurations that follow. Consequently, an initial configuration will give rise to a directed graph of configurations, call it the "configuration graph of $\mathscr{M}$", which could be infinite (e.g., if the sequence contains only non-final configurations of updates of the same agent that does not change its state and update thus never lead to a final configuration) – we will use $Cfg_{\mathscr{M}}$ to denote the graph of all configurations of $\mathscr{M}$ that successively follow any configuration from $Init_{\mathscr{M}}$.

While we are, in general, interested in the shortest path through the $Cfg_{\mathscr{M}}$ since that path will give us the desired result (i.e., the terminal configuration(s) we are interested in), the shortest path may not be unique. And, moreover, it is possible that there are two shortest paths that result in *different terminal conditions*. Standard discrete-time simulations do not usually distinguish between different terminal states that differ only with respect to the ordering of the agents in the "cycle-based update strategy" (i.e., there might be two agents in the same update cycle that cause the termination of a simulation), although this problem can be easily avoided by finishing the updates of all agents within the same cycle (e.g., by requiring as an additional condition for termination that all agents *are* at the same cycle). We will now formally define the notion of update strategy:

Definition 3 (Update strategy). *An* update strategy *or* update policy *for an S-ABM $\mathscr{M}$ is a mapping $\pi_{\mathscr{M}} : Cfg_{\mathscr{M}} \mapsto \mathscr{P}(Agents_{\mathscr{M}})$ from possible (consistent) configurations to the powerset of all agents, which effectively in each possible configuration selects a subset of agents for updating. An update strategy $\pi_{\mathscr{M}}$ is* consistent *if for all configurations $C \in Cfg_{\mathscr{M}}$ with $\pi_{\mathscr{M}}(C) = \mathscr{A}$ and all $A \in \mathscr{A}$ at cycle k, there is no agent $A' \in C$ such that A' within I_A and A' is at a different cycle $l \neq k$. A simulation $\langle C_0, C_1, \ldots, C_k \rangle$ is updated based on a* cycle-based update strategy *whenever for two configurations in sequence in the simulation all agent states are at most one cycle apart (i.e., for any two configurations in sequence C_l and C_m with $A_l \in C_l$ and $A_m \in C_m$, if A_l is at the $i-th$ cycle and A_m is at the $j-th$ cycle, then $|i-j| \leq 1$).*

It follows immediately that cycle-based update strategies (such as updating agents based on some ordering of their unique IDs) are consistent. While cycle-based update strategies are commonly used in and appropriate for discrete-event simulations on single computer systems, they do not necessarily give rise to good performance in distributed simulations, as we will see in Section 3.

As a side remark, simulations of agent-based models, as defined above, are *deterministic* and thus *reproducible* from initial configurations. However, it is sometimes desirable to allow for “non-deterministic” state transitions (e.g., to model probabilistic state transitions where each transition has a certain likelihood associated with it). The above definitions can be straightforwardly augmented to allow for non-determinism by dropping the requirement that agent updates be functions and constructing them as annotated relations instead, where the annotation is a numeric value in $[0,1]$–the transition probability–such that all annotations of transitions from a given state with the same update sum to 1.[7]

2.2 Distributing Simulations of Agent-Based Models

Spatial agent-based simulation models can be automatically parallelized and distributed in different ways. One obvious way is to run each agent on its own processor. Before an agent can update its state, it needs to collect the current state information from all other agents (running on other processors). Once the information is available, the agent updates its state and begins the update cycle again. All processors update their respective agents in the very same cycle-based fashion to ensure the correctness of the results. Another possibility is to determine in advance whether an agent needs the state of another agent for its update and to distribute agents based on

[7] The consequences for implementations are that explicit representations of random number generators and their seeds are necessary to be able to reproduce simulations. Reproducible simulation runs are then defined in terms of the seeds of the random number generators and the initial states (i.e., at any choice point the random number generator will deterministically produce a next “random number”, which is used to determine the state transition).

these dependencies (e.g., subsets of mutually dependent agents end up on the same processor, which limits the exchange of state information to exchanges among local agents). Other options are to predict or (empirically) determine the actual update time of an agent and run computationally expensive agents on separate processors, while running computationally cheap agents together on one processor.

The ideal case would be a setup where only one partition $\Pi(C_0)$ of the agents in the initial configuration C_0 has to be computed and after distributing and initializing all agents on their respective processors, each processor can update its agents independently and asynchronously until a final configuration is reached.[8] Unfortunately, this case is rarely true of agent-based models given that they are typically used for the study of interactions among agents. Hence, additional mechanisms are required to synchronize the states of agents residing in different simulations on different processors. "Synchronization" here means that if a simulation instance running on processor P_i requires the state of an agent E_j from a "remote" simulation instance running on another processor P_j, then the simulation on processor P_j needs to be able to send this information back to the simulation on processor P_i.

Which of the above approaches works best will depend on various factors, including the complexity of the update function of the involved agents, the distribution of agent types in a particular setup, the computational overhead of sending state information requests and receiving them (including network latencies), the pool of available processors (e.g., individual speeds, etc.) and whether this pool remains constant throughout a simulation run or can change over time, etc. All these factors (and their interdependencies) are important for efficient parallelizations of agent-based models.

We start by formalizing the intuitive idea of splitting up a set of agents and assigning them to processors in a given set of processors.

Definition 4 (Split of Configuration). *Let $\mathscr{M}$ be a S-ABM, C a configuration in $Cfg_{\mathscr{M}}$, and $Proc = \{P_1, P_2, \ldots, P_n\}$ a set of available processors ("processor pool"). Then a* split P^C_{Proc} *of C is a mapping $P : C \mapsto Proc$–called* agent-processor assignment*–of agents to processors P_i in $Proc$.*

Note that the agent-processor assignment does not have to be surjective as we might not need all processors in the processor pool.

Corollary 1. *A split P^C_{Proc} induces a partitioning Π_C of a configuration C into i disjoint subsets of agents Π_{C_i} in C.*

Proof. It is straightforward to check that the sets $\Pi_{C_i} := \{A|A \in C \wedge P^C_{Proc}(A) = P_i\}$ for each $P_i \in Rng(P^C_{Proc})$ form a partition of C (they are disjoint and their union is C).

Each "subconfiguration" Π_{C_i} is itself a configuration and can thus be updated in the same way as C. In the context of parallelizing a simulation, i.e., a sequence of configurations, we can simply split the initial configuration C_0 of a simulation among

[8] Note that detecting final configurations in a distributed simulation can be very tricky and will be briefly addressed in the Discussion section.

the processors in *Proc* and then continue to update the agents on each processor P_i independently as long as the states of agents updated by P_i do not depend on the states of agents updated by other processors P_j. If there is such a dependence, then there are two options: (1) either the external state information has to be obtained before the state of the local agents can be updated, or (2) both configurations are "merged" before the update (we will consider both options below).

Hence, the critical aspect in parallelizing a spatial agent-based simulation is to detect these dependencies automatically and communicate the necessary information among processors. We will first formally define the notion of "update independence", and then propose a sufficient condition for detecting it in Section 2.3.

Definition 5 (Update Independence). *An agent* A_1 *is* update-independent $UI_C(A_1,A_2)$ *of another agent* A_2 *in a configuration* C *(with* $A_1,A_2 \in C$*), if the updated state of* A_1 *in each following configuration* C' *of* C *is the same as in each respective following configuration* $(C-A_2)'$ *of* $C-A_2$ *(i.e., the configuration obtained from* C *by removing agent* A_2*).* A_1 *is called* update-dependent *on* A_2 *in* C *if* $\neg UI_C(A_1,A_2)$*.* A_1 *and* A_2 *are called* mutually update-independent *in* C *if* A_1 *is update-independent of* A_2 *and vice versa (see Figure 1). Two subconfigurations* $C_1,C_2 \subseteq C$ *are* mutually update-independent $UI_C(C_1,C_2)$ *if* $\forall A_1 \in C_1, A_2 \in C_2[UI_{C_1}(A_1,A_2) \wedge UI_{C_2}(A_2,A_1)]$*. A set of configurations* $\mathscr{C}$ *is* update-independent *if*

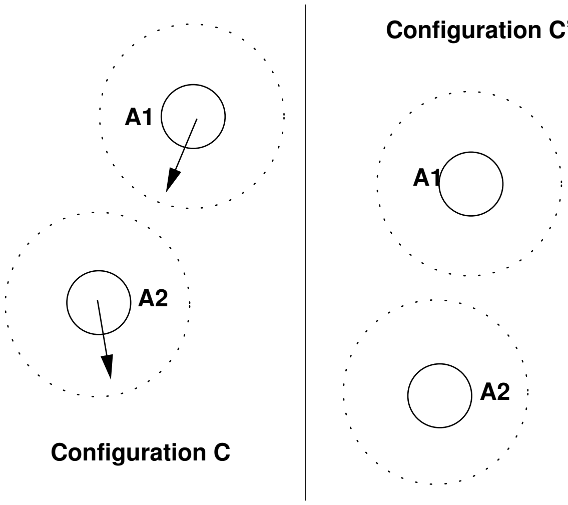


Fig. 1. An illustration of "update independence". Two agents A_1 and A_2 are both about to move in different directions (as indicated by arrows) in a configuration C. Since their interaction ranges (indicated by dashed circles) within which they can affect their environment do not overlap, either agent can be removed in C and will end up in the same position in C' (on the right) if the reduced configuration is updated as when C is updated with both agents. Hence, A_1 and A_2 are mutually update-independent.

$\forall C_1, C_2 \in \mathscr{C} UI_C(C_1, C_2)$. A split P^C_{Proc} is update-independent if the set of all Π_{C_i} is update-independent.

In other words, the presence of the other agent A_i cannot have any effect on A if its removal does not change the update of A. Note that update-independence is *not symmetric* (that is why we need the additional notion of "mutual update-independence"): it is possible that one agent A_1 is update-independent in C from another agent A_2, while the latter is not update-independent in C from the former (e.g., consider A_1 with interaction range of 10 located in (0,0) and A_2 located in (0,50) with interaction range 100 for its sensors only; then A_2 can sense A_1 and might change its behavior based on the perception without being able to affect A_1, while A_1 is oblivious to A_2's presence). Moreover, update independence is not transitive either for obvious reasons, nor is it reflexive (e.g., an agent's behaviors might or might not be completely independent of its own state).

Most importantly in the present context, update-independent configurations have the nice property that they can be directly "merged":

Corollary 2. *Let C be a configuration and Π_{C_i} update-independent configurations obtained by splitting C via P^C_{Proc}. Then $update(C) = \bigcup update(\Pi_{C_i})$ (where $update()$ is applied to all agents in the configuration).*

Proof. By induction on the size of the split. The base case, $C = P^C_{Proc}$ is obvious. Assume the Corollary has been shown for splits of size n. Then observe that for splits of size $n+1$, $update(C) = update(P^C_{Proc}) = update(\bigcup \Pi_{C_i}) = update(\bigcup^{-(n+1)} \Pi_{C_i} \cup \Pi_{C_{n+1}})$ where $\bigcup^{-(n+1)}$ is the union over the first n configurations. By induction assumption, it follows that $update(\bigcup^{-(n+1)})$ is the same as $\bigcup update(\Pi_{C-(n+1)})$, the union of the updates of all configurations Π_{C_i} except for $\Pi_{C_{n+1}}$. Now observe that $update(\Pi_{C-(n+1)} \cup \Pi_{C_{n+1}}) = update(\Pi_{C-(n+1)}) \cup update(\Pi_{C_{n+1}})$ given that the update of an agent $A \in \Pi_{C-(n+1)}$ does not depend on any agent in $\Pi_{C_{n+1}}$ since A is update-independent from all agents in $\Pi_{C_{n+1}}$ (by def. of update-independence of two configurations). The analogous argument shows that is is also true for all agents in $\Pi_{C_{n+1}}$. Hence, $update(C) = \bigcup update(\Pi_{C_i})$.

The fact that update-independent configurations can be directly merged suggests a straightforward way to parallelize a given agent-based simulation with initial configuration C_0:

```
P-ABM_G ($C_0$, $Proc$, $\mathscr{M}$) $C := C_0$
while $\neg\exists f \in Term_{\mathscr{M}} : f(C) = true$ do
    compute an update-independent split $P^C_{Proc}$ of $C$ for $Proc$
    distribute each subconfiguration $\Pi_{C_i}$ onto $P_i$ in $Proc$
    compute $update(\Pi_{C_i})$ on each $P_i$ and merge all $\Pi_{C_i}$ into $C$
    $Proc := update(Proc)$
end while
```

It is a direct consequence of merging at the end of each update that the algorithm[9] is "step-wise correct" in the following sense:

Definition 6 (Step-wise Correctness). *Let $\mathscr{A}$ be a parallel algorithm for updating a spatial agent-based simulation $S_{\mathscr{M},C_0} = \langle C_0, C_1, C_2, \ldots, C_{final} \rangle$ of an S-ABM $\mathscr{M}$. $\mathscr{A}$ is* stepwise correct *if it produces a sequence of split configurations $\langle \Pi_{C_0}, \Pi_{C_1}, \Pi_{C_2}, \ldots, \Pi_{C_{final}} \rangle$ such that $C_k = \bigcup(\Pi_{C_k})$ for all $0 \leq k \leq final$.*

Corollary 3. P-ABM$_G$ *is step-wise correct.*

Note that the above algorithm is *adaptive* because the set of available processors is updated after every configuration update. Hence the algorithm can take the new set of resources (e.g., a larger number of available processors) into account when the new split is computed.

Aside from the question of how to compute an update-independent split, to which we will return shortly, it is clear that a parallelization of a simulation according to the above algorithm is only worthwhile if the cost of computing such a split, distributing subconfigurations and merging them subsequently is low compared to the cost of updating agents. At the same time, if updating an agent is very expensive, splitting agents based on update-independence might not be the best option in the first place. For example, if C consists of a large subconfiguration C_i of update-dependent agents, this configuration will be updated on one processor and thus incurs a computation cost linear in $|C_i|$, which is in the worst case $\mathscr{O}(|C|)$. In such a case it is likely better to further split agents in $|C_i|$, distribute them over different processors, and use a mechanism to request and transfer the states of update-dependent agents in other subconfigurations as part of the update of an agent on-demand.

2.3 Towards Exploiting Properties of Spatial ABMs

As already mentioned, the important missing ingredient that is needed to be able to implement parallel algorithms like the above is an *efficient* way to detect update-independence. Detecting update-independence directly based on the definition of update-independence clearly defeats the purpose. In order to determine whether a split is update-independent for a given pool of processors *Proc* would require repeated computation of a split, independent update of all subconfigurations, and then a comparison of the merged updated subconfigurations to the update of the whole configuration. This means that the computational cost (in terms of space and time)

[9] Note that **P-ABM$_G$** is an *algorithm* because it is always possible to compute a (trivial) update-independent split in the following inefficient way: choose a split (at random), run the simulation in parallel for one step, and then compare the result to the simulation updated without a split (i.e., run on a single processor): if the simulation states are the same, then the split was update-independent (repeat for all permutations). While this way of computing an update-independent split obviously defeats the purpose of parallelizing a model in the first place (as the whole simulation needs to be updated without being split), it shows that there is always a way of computing it, hence **P-ABM$_G$** is an algorithm.

of parallelizing and updating the subconfigurations in parallel is higher than computing the update of the whole configuration at once.

Fortunately, in spatial ABMs there is another criterion that is sufficient (but not necessary) for detecting the update-independence of two agents: being within each others' interaction range. For, clearly, agents that are *not* within interaction range of each other in a given configuration cannot possibly have any effect on each other, by definition, and are thus mutually update-independent.

Note that being outside of each other's interaction range is a "conservative" estimate for mutual update independence, because two agents can still be update-independent even if they can sense each other (either because they do not take perceptions of each other into account or because their perceptions coincidentally do not have any influence on the update in the particular context). In some cases, a finer-grained distinction may be possible and desirable (e.g., when a type of agent always ignores perceptions of its own kind). The general difficulty connected to any better derivation of potential interactions, however, is how to determine them automatically from the agent update functions, which may not be possible in a practical implementation if their representations are not explicitly accessible (and even then, this will, in general, only be possible in a limited way).

Since each agent A, as part of its state, contains a translation function T_A which determines for a given location the maximum distance the agent can travel within one update cycle, the set of locations that A can influence after k of its update cycles is given in terms of T_A^k (i.e., applying T_A repeatedly up to k times to each location in the set of locations returned after each application). In a continuous 2D metric environment with $T_A > 0$, this will be the radius of an expanding circular region (as T_A^k amounts to all locations within $k \cdot T_A$). Call this expanding subspace of the environment that results from the motion of an agent starting in a given configuration C_i the agent's "event horizon":[10]

Definition 7 (Event Horizon). *The* event horizon $EH(A, C_i, k)$ *of an agent A starting in configuration C_i is the set of all locations T_A^k after k updates based on its location in C_i.*

Clearly, the event horizon of agents A in metric environments with $T_A > 0$ is monotonically increasing, symmetric, reflexive, but not transitive (which is important for computing dependencies among agents). Figure 2 shows the expanding event horizon in a metric 2D environment where T_A models the "maximum speed of locomotion" of A.

We can now refine the above algorithm for S-ABMs by merging only those subconfigurations that have update-dependencies across updates. The others can

[10] The term *event horizon* has been previously used in a slightly different sense in the domain of parallel simulation. E.g., "event horizon" in [11] refers to the set of events $\mathscr{E}$ that can occur before the first consequent event E' generated by an event $E \in \mathscr{E}$. Hence, it is the set of events $\mathscr{E}$ that can be safely executed in parallel, because no effects of any events in that set are seen during that time frame. This is similar to the way the term is used above, however, our usage refers to the first cycle an agent *could* affect another agent, rather than when it *will*.

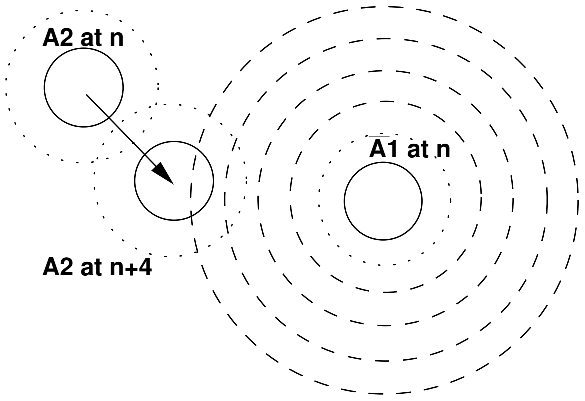


Fig. 2. An illustration of the event horizon. Agent A_2 moves from its position at cycle n to the new position at cycle $n+4$ (indicated by the arrow). The position of the agent represented by proxy agent $\overline{A_1}$ at cycle n is known, but not thereafter. The dashed circles indicate the increasing event horizon of that agent for subsequent cycles (including the maximum of the two sensory ranges–sensory ranges are indicated by dotted circles). At cycle $n+4$ A_2 intersects with the event horizon of $\overline{A_1}$ indicating that the actual position of A_1 is required before the update of A_2 can be computed.

continue to update without merging. To determine which subconfigurations need to be merged and which can continue, we introduce the notion of a "proxy agent", which serves as a (local) placeholder in a subconfiguration for the last known state of an agent updated in another subconfiguration (on another processor).

Definition 8 (Proxy agent). *A* proxy agent $\overline{A}$ *of an agent A (in the following always denoted by a bar) consists of the agent's state with the location* L_A *replaced by a set of possible locations* $\overline{L_A}$ *and the update function* U_A *replaced by* $\overline{U_A}$ *(the function that just repeatedly applies* $T(A)$ *to* $\overline{L_A}$ *on each cycle).*[11]

Proxy agents merely have a *representational function* and cannot be updated like regular non-proxy agents (i.e., they cannot change their state across configurations). Yet, they are used to compute the event horizon of the agent in subsequent configurations based on the last known configuration at which the proxy agent was updated by repeatedly applying $\overline{U_A}^k$ to each $L \in \overline{L_A}$. That way, given the state of a proxy-agent $\overline{A_j}$ (representing agent A_j in subconfiguration C_j) it is is possible to determine the subspace of the environment on which an agent A_j in configuration C_i could exert any influence in subsequent updates of C_i and thus the number of updates of C_i (based on the known states and state changes of agents in C_i) before any interaction between A_j and any $A_i \in C_i$ is possible.

We can now state an important lemma (for a proof, see [7]):

Lemma 1 (Interaction Lemma). *Let* C_1 *and* C_2 *be two subconfigurations of a configuration C containing only non-proxy agents and let* C_1^* *and* C_2^* *be the*

[11] We will extend the bar notion of proxy agents to sets of proxy agents (e.g., if *Agents* is a set of agents, then $\overline{Agents}$ is a set of proxy agents obtained from the agents in *Agents*).

configurations obtained from C_1 and C_2 by adding the proxy agents in $\overline{Agents_2}$ and $\overline{Agents_1}$ that represent the states of some non-proxy agents in C_2 and C_1, respectively. Moreover, let n be the largest number such that no non-proxy agent $A_1 \in C_1$ has $L_{A_1} \in EH(\overline{A_2}, C_1, n)$ for any $\overline{A_2} \in \overline{Ent_2}$ and no non-proxy agent $A_2 \in C_2$ has $L_{A_2} \in EH(\overline{A_1}, C_2, n)$ for any $\overline{A_1} \in \overline{Ent_1}$. Then for all $k \leq n$, $U^n_{\mathscr{M}}(C_1) \cup U^n_{\mathscr{M}}(C_2) = U^n_{\mathscr{M}}(C_1 \cup C_2)$. Or put differently, C_1 and C_2 are mutually update-independent for at least the first n updates.

The *Interaction Lemma* confirms that two mutually update-independent sub-configurations C_1 and C_2 can be updated independently as long as none of the event horizons of the proxy agents in either configuration contains a location of a non-proxy agent in that configuration. When such a configuration is reached, the actual state of the agent represented by the proxy agent needs to be obtained. Hence, we can formulate the following refined version of **P-ABM**$_G$ for S-ABMs:

```
**P-ABM**$_S$ $(C_0, Proc, \mathscr{M})$
$oldProc := \emptyset$
$k := 0$
**while** $\neg\exists f \in Term_{\mathscr{M}} : f(C_k) = true$ **do**
    **if** $oldProc \neq Proc$ **then**
        compute an update-independent split $P^{C_k}_{Proc}$ for $Proc$
        distribute each configuration $\Pi_{C_{k,i}}$ onto $P_i$ in $Proc$
        $\Pi^*_{C_{k,i}} := \{\overline{\Pi_{C_{k,j}}} | \Pi_{C_{k,j}} \in P^{C_k}_{Proc} \wedge i \neq j\} \cup \{\Pi_{C_{k,i}}\}$
        $oldProc := Proc$
    **end if**
    compute all $EH(\overline{\Pi_{C_j}}, C, k)$ for the last known state from some configuration $C$
    **for** proxy agent $A_j$ that has a non-proxy agent $A$ within $EH(\overline{\Pi_{C_j}}, C_k, k)$ **do**
        get state of $A_j$ at $k$ from $processor_j$ and update $\overline{E_j}$
    **end for**
    compute $(\Pi^*_{C_{k,i}})$' := $update(\Pi^*_{C_{k,i}})$ on each $processor_i$
    update $Proc$
    **if** $oldProc \neq Proc$ **then**
        merge all $C_{k+1} := \bigcup U_{\mathscr{M}}(\Pi_{C_{k,i}})$
    **end if**
    $k := k + 1$
**end while**
```

It follows that **P-ABM**$_S$ is step-wise correct (see [7] for a proof sketch).

3 Update Strategies for Distributed Parallel Agent-Based Simulations

The main advantage of **P-ABM**$_S$ over **P-ABM**$_G$ is that it does not require all simulation instances running on different processors to synchronize after all agents in each of the distributed simulation instances have been updated once. Rather, as long as all non-proxy agents located in a given simulation instance are located outside the event horizon of all proxy agents, the simulation instance can update its agents without requiring information from any of the other simulation instances. On the other hand, when there are potential interactions, as determined by the proxy agents' event horizons, simulations do not necessarily have to be merged, instead it suffices to update only the proxy agents based on the communicated locations of the non-local agents (in other simulation instances) they represent. Consequently, it is also not necessary to compute new update-independent splits before every update cycle (although simulations will still have to be merged and splits will still have to be recomputed, as with **P-ABM**$_G$, should the processor pool *Proc* change to preserve the adaptiveness in **P-ABM**$_S$). To elucidate how this asynchronous update could work, we start with an intuitive example, and then look at the properties of asynchronous updates more formally.

Suppose $agent_{15,4}$, i.e., the agent with $ID = 15$ in simulation instance 4, requires at its cycle 321 an update for its proxy agent $proxy\_agent_{64,7}$ (i.e., the proxy agent representing the agent with $ID = 64$ in simulation instance 7). Furthermore, let's assume that both simulation instances, 4 and 7, have been running asynchronously up to that point without communicating with each other. When simulation instance 7 gets the request from simulation instance 4 to send the pertinent information (i.e., the reduced state) of $agent_{64,7}$, $agent_{64,7}$ is already at cycle 598 (in simulation instance 7 due to the asynchronous updates). At first glance, the mismatch in cycle numbers seems to prevent an information transfer that can be used in a way that will keep the distributed simulation consistent. On further examination, however, it turns out that it is completely unproblematic for simulation instance 7 to communicate the current reduced state of $agent_{64,7}$ and for simulation instance 4 to use it (instead of the state of $agent_{64,7}$ at cycle 321) – why is that? The answer lies in the event horizon of proxy agent $proxy\_agent_{15,4}$ in simulation instance 7, which represents $agent_{15,4}$ from simulation instance 4: if there had been any chance for $agent_{15,4}$ to interact with agent $agent_{64,7}$ before cycle 598, then $agent_{64,7}$ would have ended up being located within the event horizon of $proxy\_agent_{15,4}$ in simulation instance 7 before that cycle and simulation instance 7 would have requested an update (i.e., reduced state) from $agent_{15,4}$ in stimulation instance 4. However, since no such request occurred based on our assumption, $agent_{64,7}$ never ended up being located within the event horizon of $proxy\_agent_{15,4}$ and therefore never had a chance to interact with $agent_{15,4}$, at least until cycle 598. Since there cannot be any earlier interaction, simulation instance 4 can simply use the reduced state of $agent_{64,7}$ at cycle 598 and set its proxy agent $proxy\_agent_{64,7}$ to that state, and even skip updating the agent's event horizon until all other agents reach cycle 598.

We formally summarize the above argument in a proposition:

Proposition 1. *Let $A_{i,m}$ and $A_{j,n}$ be agents with agent IDs i and j, respectively, and let S_m and S_n be two simulation instances with $A_{i,m} \in S_m$ at cycle m and $A_{j,n} \in S_n$ at cycle n, with $m < n$ such that for all cycles $m \le k \le n$ $A_{j,n}$, is not in the event horizon of $\overline{A_{i,k}}$ (where $\overline{A_{i,k}}$ is the proxy of agent $A_{i,k}$ in S_n). Then for all cycles $m \le k \le n$, $A_{i,k}$ is not in the interaction range of $A_{j,k}$ and vice versa.*

Proof. Suppose there is a cycle l such that $m \leq l < n$ at which both agents are within interaction range and suppose further that cycle $c \leq m$ was the last time that simulation instance S_n updated its proxy agent $\overline{A_{i,l}}$ based on the actual state of $A_{i,l}$. Since the event horizon of is the maximum range at any given cycle within which an agent can interact and for no cycle c with $j \leq c \leq n$ was $A_{j,c}$ within the event horizon of $\overline{A_{i,c}}$, by deviation of $\overline{A_{i,c}}$, $A_{i,c}$ could not have interacted with $A_{j,c}$. Contradiction.

Intuitively, it seems clear that the ability of simulation instances to run asynchronously and only communicate agent states when necessary should lead to performance improvements, and we have indeed been able to show previously that running simulations asynchronously using the above proposition leads to better performance than running simulations in lock-step (as is required for **P-ABM**$_G$) [7]. However, the extent of the performance improvement depends on several factors, including the complexity of the agent update function and the distribution of the agents across simulation environments, but most importantly on the update strategy *given agent update functions and distributions*. Unfortunately, there is no *general update strategy* that will yield optimal results, i.e., maximum parallelism among distributed simulation instances.

To see this, consider two simulation instances with two agents each as arranged in the left part of Figure 3. Agents A and D are non-proxy agents in simulation instance 1 and proxy agents in simulation instance 2, and, conversely, agents B and C are non-proxy agents in simulation instance 2 and proxy agents in simulation instance 1. Each agent only moves in the direction indicated by the arrow pointing away from its center. The dashed circles indicate the agents' interaction ranges. We further assume that the maximum change in location that agents can perform in one update cycle is large enough so that agent D will be in the event horizon of proxy agent B in simulation instance 1 and agent C will be in the event horizon of proxy agent A in simulation instance 2, hence requiring both simulation instances to get the updated states of the non-proxy agents from the other simulation instance. In fact, an update strategy that decides to update agents C and D first, can lead to a lock-step process where on every cycle updates for proxy agents are required, which will, in turn, trigger updates for the remaining agents. As a result, a "cycle-based update strategy" (which is the default in many simulation environments) is not a good choice for the given scenario because it requires updates on every cycle and forces both simulation instances to be in sync at each cycle, thus effectively forcing the distributed simulation to run in lock step.

If, on the other hand, simulation instance 1 updated agent A and simulation instance 2 updated agent B a few times before updating agent C, then the communicated state information would show that agent A has moved out of the way and that agent C could move freely for a certain number of cycles until it has the same cycle number as proxy agent A. In fact, the best update strategy for a situation where a simulation has to be run for a fixed number of cycles n is to first update agents A and B for n cycles and then update agents C and D for n cycles. This is possible because agent A will never be in the interaction range of any agent in simulation instance 1, and agent B will never be in the interaction range of any agent in simulation instance 2. Hence, they are update-independent and can be updated until the terminating condition is reached. As a result, this update strategy will require only one communication of the updated states of *A* and *B* (namely after the first update of *C* and *D*), then *C* and *D* too can be run to completion for $n-1$ cycles – note that at least one update of state information is necessary given the initial condition, hence this update strategy is optimal.

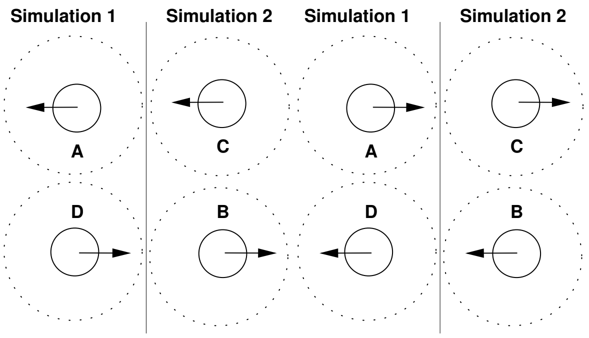


Fig. 3. Two simple scenarios demonstrating that a general optimal update strategy does not exist (see text for details).

While is it possible to have optimal update strategies for particular scenarios such as the above, it is now also easy to see that there cannot be a general optimal update strategy that is best for in every scenario. Consider the right part of Figure 3, which is the same setup as on the left except that all agents move in opposite directions. Hence, the optimal update strategy is to "reverse" the update sequence from before, starting with updating agents *C* and *D* for *n* cycles, followed by updating *A* and *B* for *n* cycles (with one state update required for each proxy agent after cycle 1). Since both scenarios are the same – same number of agents, same split – yet the optimal strategy is different for each scenario, there cannot be a general algorithm that determines the optimal strategy based on the number of agents and splits alone. Rather, the direction of movement, which is determined by the agent update function, is essential for selecting the best update strategy, hence:

Fact 1. *There is no general algorithm which for any given split of a spatial agent-based model simulation can determine the best update strategies for each simulation instance (without knowledge of the specific agent update function).*

Even though there is no general algorithm to determine optimal update strategies, it still makes sense to attempt to define heuristics that will improve over the above lock-step behavior caused by the "cycle-based update strategy". In the following, we will discuss four proposals of such strategies and later show how three of them can be an improvement over the "cycle-based update strategy" in an agent-based model taken from a real-world modeling application.

3.1 A General Alternative Scheduling Strategy for Agent-Based Models

Alternative scheduling strategies for ABMs can be used to avoid some of the inefficiencies associated with "cycle-based update strategies". Unlike the typical "cycle-based update strategy", which require repeatedly updating every agent in a given configuration once (without updating one agent twice unless all other agents have been updated at least once), alternative scheduling strategies relax this update constraint. Instead, they allow for agents to be at different cycles within one simulation instance as long as these cycle differences do not lead to inconsistent update sequences.

In a way, alternative scheduling strategies apply the idea of **P-ABM**$_S$, that parallel simulation instances can run asynchronously as long as none of their agents are within each others' event horizon, at the level of an individual simulation instance: all agents in a subset $AS \subseteq C$ of a configuration C can be updated until one of the agents $A \in AS$ enters the event horizon of some agent $B \notin AS$. Note that for this idea to work, we have to extend our notion of proxy agent to agents *within the same simulation instance*. Specifically, we have to first select a subset of agents $AS = \pi(C)$ based on our selection policy π given the current configuration C such that all agents $A \in AS$ are at the same cycle n. Then we replace the remaining set of agents $RS := C - AS$ with proxy agents that will get updated along with the selected non-proxy agents. This is done to detect possible interactions with agents outside of AS at which point AS is no longer update-independent. To make this update strategy work in a consistent fashion, it is important to pay attention to the cycle at which each agent $B \in RS$ is when their respective proxy agent is initialized: if B is at a cycle $\leq n$, then the event horizon of its proxy agent $\overline{B}$ has to be computed for cycle n; otherwise the proxy agent will not be updated until its cycle number $> n$ is reached. We can summarize this scheme as the general alternative scheduling algorithm **AltSched**$_G$:

```
**AltSched**$_G$ $(C_0, \pi, \mathscr{M})$
$C := C_0$
**while** $\neg\exists f \in Term_{\mathscr{M}} : f(C) = true$ **do**
    $AS := \pi(C)$ (with all $A \in AS$ at the same cycle $k$)
    $RS := C - AS$ (*)
    $\overline{RS} := \{\overline{B} | B \in RS\}$
    $C := (C - RS) \cup \overline{RS}$
    **while** $\neg\exists A \in AS, B \in RS : L_A \in EH(B, C, cyc(A)) \wedge \neg\exists f \in Term_{\mathscr{M}} : f(C) = true \wedge$
    $cycle(AS) < k + maxupdate$ **do**
        $C := update(AS \cup \overline{RS})$
    **end while**
    $C := (C - \overline{RS}) \cup RS$ (**)
**end while**
```

There are several important points to notice about **AltSched**$_G$. First note that **AltSched**$_G$ is consistent (which follows from the Interaction Lemma) but will in general not lead to the same simulations as cycle-based strategies. While the latter are guaranteed to find a termination condition (if it exists) given that they perform a breath-first search, whether **AltSched**$_G$ will find a terminal condition critically depends on how the update policy π selects subsets of agents and on how long they are updated using *maxupdate* (the maximum number of updates to be performed on a set AS before another set is chosen again based on π). In infinite configuration graphs, for example, a depth-first update strategy might fail to find a terminating condition. A simple solution to this problem is to require of π that no agents in a configuration C be selected that are more than a maximum difference δ cycles ahead of any other agent in C.

Second, **AltSched**$_G$ subsumes the standard "cycle-based update strategy" using $maxupdate = 1$ and the policy $\pi(C) = C$ for all C in a model $\mathscr{M}$.

Third, with only minor modifications it lends itself to multiple asynchronous parallel runs: simply recursively apply π to RS at the line marked (*) yielding a set of agents AS_i and their associated proxy agents RS_i until $RS_i = \emptyset$, and then merge all updated non-proxy agent sets AS_i to obtain the new configuration (instead of replacing the proxy agents $\overline{RS}$ with their non-updated counter parts RS in the sequential version). This way of parallelizing a single simulation instance might be preferable over the non-parallelized version even though the parallelization incurs a small computational overhead because it will be able to automatically utilize real parallelism available on multi-processor and multi-core machines as well as idle processor time on a single processor with only one core (e.g., due to wait times on network communication in the context of **P-ABM**$_S$, or various OS blocking).

Fourth, by being able to change the update sequence of agents, using **AltSched**$_G$ can lead to much shorter simulation runs if the update policy π is sensitive to terminal conditions. For example, suppose in a simulation with 1000 agents the goal is for at least one agent to reach a particular goal location in the environment and the terminal condition is thus defined by one of the agents being in that location. Moreover, suppose that the update policy π gives priority to agents that are close to the goal location and that in the initial configuration a group of 10 agents which is outside of the interaction range of other agents is headed directly towards the goal location. To keep things simple, let us also assume that all agents travel at the same speed. Repeatedly selecting these 10 agents for update will then cause the simulation to reach a terminal state without ever having to update any of the other agents (because they will never be in the event horizon of any of the other 990 agents). As a result, the run time under the "goal-sensitive" policy π is about 1% of the runtime of the policy corresponding to the "cycle-based update strategy".

And finally, **AltSched**$_G$ can be combined with **P-ABM**$_S$ and should lead to a performance improvement for distributed simulations if general update policies can be defined that they are sensitive to the update requirements of distributed simulation instances, which we will address next.

3.2 Combining AltSched$_G$ *and* P-ABM$_S$

The goal of combining **AltSched**$_G$ and **P-ABM**$_S$ is to significantly reduce the overall runtime of parallel distributed simulations compared to the standard "cycle-based update strategy". Hence, we need to define general policies for $\mathbf{AltSched}_G$ that will select agent groups for updates in a way that better exploits the parallelism of distributed simulations in $\mathbf{P\text{-}ABM}_S$. One of the main performance reducing factors that we have seen in our previous implementation of $\mathbf{P\text{-}ABM}_S$ (with the cycle-based update strategy run by all simulation schedulers) is that simulation instances are *blocked*, i.e., that they cannot update any of their agents without first having obtained updated information on one of their proxy agents. Blocked simulations lead to idle processor time and cannot continue to utilize their computational resources until they get the requested updates. Hence, time wasted due to blocking can be reduced by any policy that is able to anticipate blocks in a remote simulation instance and update its local agents in such a way that the update information is available when requested by the remote simulation instance. Note, however, that giving preference to agent groups that will reduce remote blocking can be in tension with the goal of selecting local agent groups (within each simulation instance) that are likely to make the most progress towards reaching a terminal condition – while we briefly return to this problem in the Discussion section, we will focus here on how we can reduce the latencies and delays in running distributed simulations introduced by remote blocks.

While it is in general not possible for a given simulation instance to detect whether and when a remote simulation will require information about one of its local agents, it is possible to compute conservative estimates that if executed by all simulation instances will improve overall system performance. For example, we can determine for each local agent the earliest cycle at which the local agent could be in the event horizon of some proxy (i.e., remote) agent. An update strategy could then decide to give preference to updates of those agents, which will cause the local simulation instance to block earlier than it otherwise would have had it updated other agents first. The benefit of such "early blocks" is that the remote simulation instance can get the updated state from the blocked agent as part of the blocked simulation's request for an update on the proxy agent. Since the remote simulation instance uses the same update policy and thus also gives preference to agents that are likely to need update information in the near future, it is probable that it will either already or at least soon have an update available. Hence, if all simulation instances give preference to updates of agents whose state information will be requested by remote simulation instances in the future, the overall effect is that cycles with blocking agents will occur more frequently in the beginning of a simulation sequence compared to the standard cycle-based updates. As a result, simulation instances will likely still be able to update some of their (non-blocked) local agents while they are waiting for state updates for blocked agents as opposed to a simulation instance waiting to update any agents until the state information is received for all blocked agents. In sum, policies that are sensitive to the information demands of remote simulation instances will in many cases be able to reduce the idle time of parallel simulation instances, which in turn will lead to overall shorter simulation runs (everything else being equal).

We now define four general policies that are intended to reduce the overhead associated with blocking simulation instances in a distributed simulation based on **P-ABM**$_S$.

Remote Event First

The *Remote Event First* policy projects out the next potential "event" with an agent on a remote host (i.e., a local agent ending up in the event horizon of a remote agent). *Remote Event First* intuitively has benefits because it increases the number of agents that can be run at any one time in every simulation instance and reduces the risk that the simulation instance is completely blocked waiting for state updates from remote simulation instances. This method of ordering performs a depth-first traversal through the configuration graph and therefore is not guaranteed to terminate. Variations of *Remote Event First* (as mentioned above) can be implemented utilizing mechanisms that disallow an agent to advance too far into the future without catching up other younger agents in the same simulation instance.

Remote Blocks Then Remote Event First

The *Remote Blocks Then Remote Event First* policy is a cooperative variation of the *Remote Event First* policy that works with the other nodes to identify which agents to run next. This policy can drastically improve performance because it will quickly unblock a remote agent that is traversing though the simulation timeline. When an agent becomes blocked, that simulation instance shares this information with all of the other simulation instances causing them to give immediate priority to those agents whose updates will allow the blocked agent to progress. If some of the selected agents are also blocked, a simulation instance will revert back to the *Remote Event First* criteria.

Youngest First

A *Youngest First* policy simply chooses the agent with the lowest cycle time from all of the potentially update independent agents. This method of ordering has a breadth-first type of traversal through the configuration graph. The main benefit of such a selection policy is that it is guaranteed to terminate if an exit criterion is reachable.

Remote Blocks Then Youngest First

The *Remote Blocks Then Youngest First policy* is a variation of the *Youngest First* policy that cooperatively works with the other nodes to identify which agents to run next as described above. As with the *Remote Blocks Then Remote Event First* policy, it will revert to its base strategy of *Youngest First* when it cannot advance agents whose updates are requested by remote simulation instances because they are all blocked.

4 Implementation of P-ABM$_S$ and Experimental Evaluation

We implemented all proposed scheduling algorithms in our agent-based SWAGES environment in order to provide a (non-optimized) proof-of-concept system that tightly integrates the scheduler in the simulation environment with the parallelization and distribution algorithm. To be able to demonstrate performance gains of the proposed update strategies over the standard “cycle-based update strategy” using a practical example, we selected an actual agent-based model from a biological modeling research domain of female choice [5]. We first discuss the details of our implementation and then report the results from the empirical evaluations.

4.1 *Implementation of* P-ABM$_S$ *in* SWAGES

SWAGES is a JAVA-based agent-based simulation and experimentation server intended for any kind of computing environment (e.g., from homogeneous Beowulf clusters to heterogeneous computers connected only via the Internet). It consists of several distributed components that cooperate closely to achieve high resource utilization in a heterogeneous dynamically changing computing environment. SWAGES was used and extended to support the scheduling and monitoring of the execution of simulations for both cycle-based and non-cycle-based update strategies for agent based simulation experiments.

Without modification, SWAGES provides the communication infrastructure to start, run, and supervise simulations. It also gathers and stores simulation results in an easily accessible manner for future statistical analysis. The server can schedule sets of simulation experiments (e.g., simulations with a variety of different initial conditions) and ensure their timely completion by monitoring their performance and detecting problems with the execution (e.g., because the load on a host is too high, or the simulation crashed), in which case it can take any number of recovery actions (from resuming a simulation on a different host if its state was saved, to restarting it anew if no state information was available). Each simulation instance can run on its own host and maintains a socket connection instance for all communication purposes (e.g., information about the current simulation cycle, simulation parameters, etc. will be delivered on this connection).

SWAGES required several modifications to be able to implement and work with non-cycle-update strategies. SWAGES was extended to support the merging of distributed configuration and distributing simulation environments containing only subsets of agents across a series of processors. Associated communication support protocols were also added to provide a mechanism for simulation instances running on different processors to access the new features. In order to facilitate non-cycle-based update strategies, a single simulation instance must distribute the agents in addition to the context in which they are to run (e.g., environmental specification, agent initial conditions, agent models). This distribution could be performed inside the centralized server if the server had explicit knowledge of how to initialize an agent.

Since SWAGES can only start simulation instances, but cannot initialize agents within a simulation instance (or perform any other operation *within* simulation instances), parallelizing and distributing new simulation instances is therefore a three-step process. First, SWAGES informs a simulation instance that a resource for distributed parallel simulation is available (i.e., that there is a host computer where a new simulation instance can be run). If the simulation instance decides that it wants to split off a subset of its agents and run it on another host, it accepts the resource offer by sending back a serialized representation of all those agents that are supposed to be run in the remote simulation instance. SWAGES, in turn, launches a new simulation environment on another processor and provides it with the serialized agent set. From that point on, the local and remote simulation instances continue to update their agents, with the local instance treating the serialized agents as proxy agents and the remote instance treating all other agents as proxies. Whenever an update for a proxy agent is required by a simulation instance (because a local agent ended up in the even horizon of the proxy agent), the simulation instance will request an update from the simulation instance running the proxy agent via SWAGES.

Updated state information for any agent needs to be shared among all simulation instances. This can be done in peer-to-peer fashion using some broadcast or shared memory mechanism, or it could be accomplished using a server to broker the communication. SWAGES uses the latter mechanism and acts a global repository for the updated agent states computed in the simulation so that proxy representation can be updated on demand. The updating of a simulation instance and request for proxy agent state is accomplished in a non-blocking manner to allow agents in other instances of the algorithm to be chosen and updated during the slow I/O operation of communicating the updates. During the communication phase newly generated agent states are shared and received. The information of new update states of remote agents is stored in their respective proxy representation to be used by the simulation instance. If a requested agent state does not exist in the central repository, SWAGES will store the request until the data becomes available and also inform the simulation instance that "owns" the agent that another simulation instance requires updated state information. This information can be used to influence the update policy in **AltSched**$_G$ (e.g., in the *Remote Block* policy).

To implement the above policies and select a set of (unblocked) agents AS that can be updated, we start with a set that contains a given agent A chosen by the policy and then recursively add into A all agents within the event horizon of any agent already in the set. This final set $TC(A)$ forms a *transitive closure* of A that contains all agents that are connected via their overlapping interaction ranges and is thus update independent from all other agents in the simulation instance (i.e., agents in the set $C - AS$).[12] If AS contains a proxy agent, then AS can be updated only once until that proxy agent's state updates thus blocking all agents in AS by

[12] Another way to view this concept is to consider the collection of agents as a graph of all agents at a given time. Let each node represent an agent and directed edges represent that agent's ability to sense or influence the connecting node. All nodes that are reachable from a given node define a transitive closure. Therefore, the collection of agents in a transitive closure is a set of agents that can be updated independently from other agents in the simulation. Independent updating is possible because agents outside of the transitive closure have no ability to influence or be influenced by those agents inside the transitive closure.

placing their updates on hold. Optionally, the list of agent states that were updated is sent back to the SWAGES server for bookkeeping. If no agent states are updated, a list of proxy agents causing blocking in that simulation instance is requested. The SWAGES server responds with any updated agent states requested or previously requested by that simulation instance. Additionally, a list of identifiers of agents in that simulation instance that are causing blocking in other simulation instances is also sent, which can be used in update policies.

4.2 Evaluation

We evaluated the three most promising update strategies defined above: *Remote Event First with Remote Blocks*, *Youngest First*, and *Youngest First with Remote Blocks*.[13] In addition, we included the "Cycle-Based Update Strategy" as a standard control condition. For the simulation model, we picked a realistic model from one of our current agent-based modeling domains, that of *female mate choice in treefrogs*. In this model, male and female treefrogs are located in a swamp area. Male frogs are stationary within the environment and indicate their presence and readiness for mating by repeatedly making "mating calls" of fixed, but different quality. Females initially enter the swamp from the rim, listening to the males' calls and repeatedly making choices about which of the males to approach based on the "quality" of male's mating call. Once a female has picked a male, she will approach the male based on the directions she obtains from locating the source of the male's call ("phonotaxis"). It is known from the biological literature that females show phonotaxis toward calls of males with higher *pulse numbers* (e.g., [9, 10]). Our specific model of female choice is intended to study the influence of male and female spatial distributions on mating success of females (measured in terms of the overall male fitness) when females pursue one of two "choice strategies": a *best-of-n* strategy where they pick the male with the best quality of the closest *n* males [3], or a *minthresh* strategy where they pick the closest male whose quality is above a minimum threshold [4] (for more details on the model, see [5, 2]).

For the evaluation experiments, ten females were initially positioned based on a Gaussian function along the rim of the swamp. 27 males were initially positioned by an inverse Gaussian function in the middle of the swamp (positioning more males towards the critical rim areas, see the left part of Figure 4). The swamp was modeled by a continuous rectangular area of 10×25 meters. In comparison, frogs are 4.75cm in size and can leap up to 1.44cm in one hop. While the male update function does not change the male frogs behavior unless a female frog is within mating range, in which case the male will mate, the female update function will map perceived call qualities onto the direction towards the chosen male and cause the female to leap (at a fixed speed of 1.44cm/sec) in that direction. When a female is within mating range of a male (4cm), she will attempt to mate (regardless of whether the male was the

[13] *Remote Event First* was left out because it did not show sufficient performance gains in our evaluation scenario even though it might work well in others.

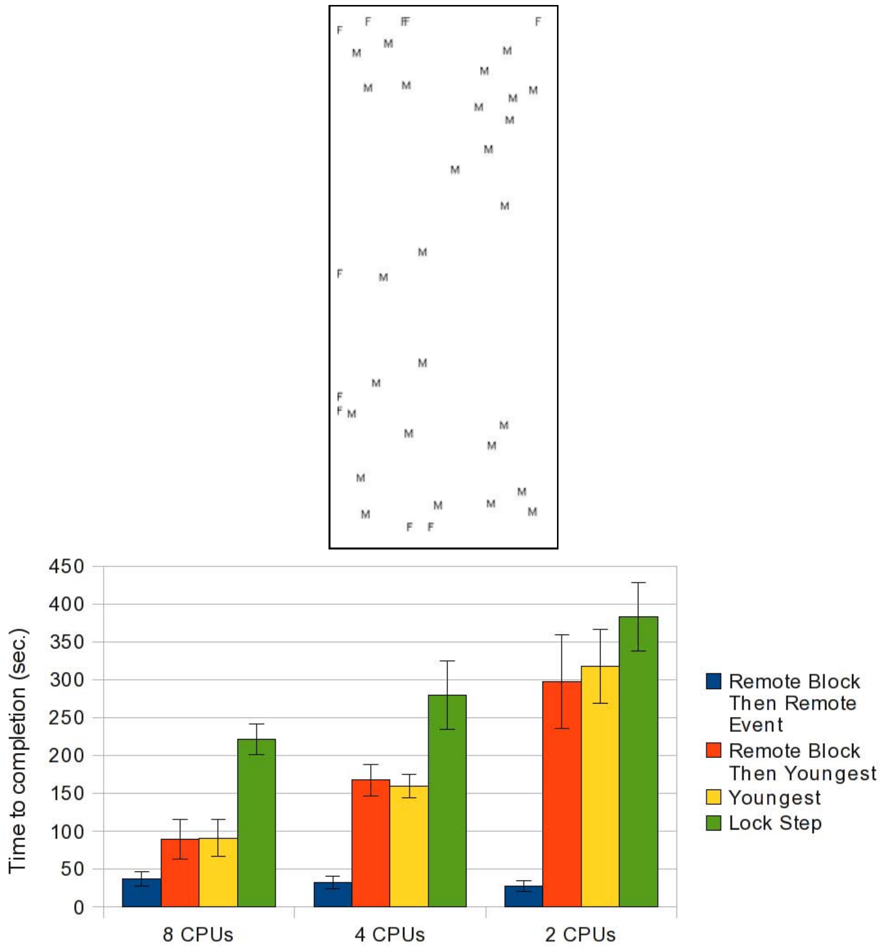


Fig. 4. The initial configuration of male and female frogs in the swamp (on the left) and the results (on the right) showing the mean overall simulation run-time (in seconds) together with their standard deviation averaged over all split conditions (see text for details).

chosen one or not, which models the biologically hypothesized behavior). A simulation run starts with placing all agents (male and female) in their initial locations and updating them until all females have mated (which is always guaranteed to happen because there are more males than females in the environment).

For the evaluation of the four update strategies, we ran the same initial configuration (keeping male and female distribution fixed as well as the distribution of the male calls) under 10 different random split conditions distributed on a fixed pool of processors (i.e., 2, 4, and 8).[14] The right graph in Figure 4 shows the results,

[14] We did not include the one processor case since there is no significant performance difference between any of the employed strategies.

which were obtained by averaging over the total simulation run-time (from starting the SWAGES gridserver with the experiment startup file until the server quit) across the different split conditions. As can be seen from the results, there is already an immediate benefit of using any of the update strategies other than the standard cycle-based strategy with more than one processor, even though the performance gain really becomes more pronounced as the number of processors increases. In particular, in the case of 8 processors, changing from the standard update strategy reduces the overall execution by more than half for all cases. Note that the excellent performance of the Remote Event First with Remote Blocks strategy has to do with the fact the agent update sequence of this policy closely aligns with the optimal agent update sequence of this simulation. The optimal agent update sequence of a simulation is based on the characteristics of the agents in the simulation as well as the simulation's terminating condition. Since males do not and females select males based on closest proximity then the projected remote event of a male-female intersection would actually accurately measures the simulations terminating condition and therefore select the agent updates necessary to achieve the simulation termination quickly. Also, the small increase in overall run-time has to do with the additional bookkeeping required for more processors, which is due to the low run-times of *Remote Event First with Remote Blocks* that shows up explicitly while being absorbed within the run-time of the other strategies (given that it is only a small fraction).

4.3 Discussion

The empirical evaluation confirms what we were expecting based on the rationale for defining our update strategies, namely that coordinating the updates of agents in distributed simulations using simple heuristics that re-order update sequences without changing the "semantics of the simulation" (i.e., the simulation outcome) can lead to significant performance improvements in the context of parallel distributed simulations. It is important, however, to keep in mind that the exact gains of using different update strategies will depend on several factors, at least on (1) the complexity of the update functions of individual agents, (2) the agent distribution, (3) the agent interaction range, (4) the agent translation function, (5) the size of the world, etc. For the above strategies, we expect to see in general the best performance gains for more complex agents where most of the processor time is spent on agent update functions relative to the simulation book-keeping. In those cases, any way that can re-order the update sequence to give priority to agents that either require information from other simulation instances or could provide information to other simulation instances (that are required at some future point) will reduce the overall simulation run-time relative to that of the "cycle-based update strategy", which is oblivious to any interactions between parallel simulation instances (recall the example from the non-optimality proof where the "cycle-based update strategy" can lead to a lock-step behavior).

It is also important to note that while the above evaluation shows significant performance improvements over the cycle-based update strategy, there is still room for further improvement. For example, it is possible to use a finer-grained distinction about dependencies among agents that does not amount to computing the full transitive closure (e.g., an agent really only needs information for agents that can potentially interact with it at cycle n and not the full transitive closure). Hence, as long as all agents in its interaction range are at cycle n, it can be updated and by keeping its previous state at cycle n around it will allow other agents in its interaction range to update at a later point without causing inconsistencies. This type of optimization can lead to fewer blocked scenarios since there will be fewer agents identified as dependent that would therefore be required to update at cycle n before any of the agents advance to cycle $n+1$.

Another interesting question is how update strategies that are intended to reduce remote blocks can be combined with heuristics that prioritize agents based on their estimated distance to a terminal condition. It is currently unclear whether there is a general answer to this question (e.g., to always prefer advancing agents close to terminal conditions if there are guarantees for the heuristic such as being *admissible*).

Finally, we would also like to point out that richer agent-based simulations that include, in addition to agents, environmental states (e.g., global or local temperature) or other entities (e.g., non-movable, but consumable food sources) will require additional mechanisms for distributing and keeping track of those state across simulations instances effectively. This is also true of locations that can have properties assigned (e.g., swamp land vs. mountain side).

5 Conclusions

In this chapter, we investigated the utility of using novel update strategies for agents in simulations of agent-based models. These strategies differ from the standard cycle-based update strategy with respect to the update sequence of agent updates from initial to terminating conditions, but without changing any simulation outcomes. We demonstrated that the performance of parallel distributed agent-based simulations extended from our previous parallelization and distribution algorithms can be significantly improved if the proposed heuristics are employed. Specifically, we were able to achieve more than 50% shorter overall simulation run times in an agent-based simulation model taken from a biological research domain that investigates female choice behavior in tree frogs.

While any particular performance improvements will always critically depend on the nature of the employed agents, the proposed heuristics seem promising across the board. This is because they attempt to anticipate information exchanges between distributed simulation instances that will likely be required at some future time and prioritize agent updates of those agents whose state will be required.

Future work will investigate how the heuristics can be adaptively combined to utilize their individual strengths. We will also investigate ways to improve the detection of update independent subsets of agents that do not solely rely on event horizons (which are only a rough estimate of possible interactions). In particular, we are interested in exploring reflection methods that will be able to gain and utilize information in the agent update function about whether an agent is likely to interact with another agent. Finally, we will also look at replacements for the transitive closure computation which is expensive and not needed in its entirety to determine subsets of agents that can be updated.

References

1. Barwise, J., Moss, L.: Vicious Circles. CSLI Lecture Notes (1996)
2. Boyd, S., Scheutz, M., Hercog, L.: Exploring female mate choice strategies in tree frogs with a spatial agent-based model (in preparation)
3. Janetos, A.C.: Strategies of female mate choice - a theoretical-analysis. Behavioral Ecology and Sociobiology 7(2), 107–112 (1980)
4. Jennions, M.D., Petrie, M.: Variation in mate choice and mating preferences: A review of causes and consequences. Biological Reviews of the Cambridge Philosophical Society 72(2), 283–327 (1997)
5. Scheutz, M.: Model-Based Approaches To Learning: Using Systems Models And Simulations To Improve Understanding And Problem Solving In Complex Domains. In: Modeling and Simulations for Learning and Instruction, Artificial Life Simulations–Discovering Agent-Based Models, vol. 4. Sense Publisher, Rotterdam (2008)
6. Scheutz, M., Madey, G., Boyd, S.: tMANS–the multi-scale agent-based networked simulation for the study of multi-scale, multi-level biological and social phenomena. In: Proceedings of Spring Simulation Multiconference (SMC 2005), Agent-Directed Simulation Symposium (2005)
7. Scheutz, M., Schermerhorn, P.: Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models. Journal of Parallel and Distributed Computing 66(8), 1037–1051 (2006)
8. Scheutz, M., Schermerhorn, P., Connaughton, R., Dingler, A.: Swages - an extendable distributed experimentation system for large-scale agent-based alife simulations. In: Proceedings of Artificial Life X (2006) (forthcoming)
9. Schwartz, J.J., Buchanan, B.W., Gerhardt, H.C.: Female mate choice in the gray treefrog (hyla versicolor) in three experimental environments. Behavioral Ecology and Sociobiology 49(6), 443–455 (2001)
10. Schwartz, J.J., Huth, K., Hutchin, T.: How long do females really listen? assessment time for female mate choice in the grey treefrog, hyla versicolor. Animal Behaviour 68, 533–540 (2004)
11. Steinman, J.S.: Discrete-event simulation and the event horizon. In: PADS 1994: Proceedings of the eighth workshop on Parallel and distributed simulation, pp. 39–49. ACM Press, New York (1994)

On the Use of Distributed Genetic Algorithms for the Tuning of Fuzzy Rule Based-Systems

Ignacio Robles, Rafael Alcalá, José M. Benítez, and Francisco Herrera

Abstract. The tuning of Fuzzy Rule-Based Systems is often applied to improve their performance as a post-processing stage once an appropriate set of fuzzy rules has been extracted. This optimization problem can become a hard one when the size of the considered system in terms of the number of variables, rules and, particularly, data samples is big. Distributed Genetic Algorithms are excellent optimization algorithms which exploit the nowadays available parallel hardware (multicore microprocessors and clusters) and could help to alleviate this growth in complexity.

In this work, we present a study on the use of the Distributed Genetic Algorithms for the tuning of Fuzzy Rule-Based Systems. To this end, we analyze the application of a specific Gradual Distributed Real-Coded Genetic Algorithm which employs eight subpopulations in a hypercube topology.

The empirical performance in solution quality and computing time is assessed by comparing its results with those from a highly effective sequential tuning algorithm. We applied both, the highly effective sequential algorithm and the distributed method, for the modeling of four well-known regression problems. The results show that the distributed approach achieves better results in terms of quality and execution time as the complexity of the problem grows.

Keywords: Genetic Fuzzy Systems, Fuzzy Rule Based-Systems, Distributed Genetic Algorithms, Genetic Tuning.

1 Introduction

Fuzzy rule based-systems (FRBS) have become a wide choice when addressing modeling and system identification problems [1, 2, 3, 4]. One of the most popular

Ignacio Robles · Rafael Alcalá · José M. Benítez · Francisco Herrera
Dept. of Computer Sciences and Artificial Intelligence,
University of Granada, 18071-Granada, Spain
e-mail: ignaciorobles@gmail.com,
{alcala,J.M.Benitez,herrera}@decsai.ugr.es

approaches for the design of FRBSs is the hybridization between fuzzy logic [5, 6] and Genetic Algorithms (GAs) [7, 8] leading to the well-known Genetic Fuzzy Systems (GFSs) [9, 10, 11]. A GFS is basically a fuzzy system augmented by a learning process based on evolutionary computation, which includes GAs, genetic programming, and evolutionary strategies, among other evolutionary algorithms [12].

The predominant type of GFS is that focused on FRBSs, since the automatic definition of FRBSs can be seen as an optimization or search problem, and GAs are a well known and widely used global search technique with the ability to explore a large search space for suitable solutions only requiring a performance measure. In addition to their ability to find near optimal solutions in complex search spaces, the generic code structure and independent performance features of GAs make them suitable candidates to incorporate a priori knowledge. In the case of FRBSs, this a priori knowledge may be in the form of linguistic variables [13], fuzzy membership function (MF) parameters, fuzzy rules, number of rules, etc. These capabilities extended the use of GAs in the development of a wide range of approaches for designing FRBSs over the last few years.

In this framework, a widely-used technique to enhance the performance of FRBSs is the genetic tuning of MFs [14, 15, 16, 17, 18, 19]. It consists of improving a previous definition of the Data Base (DB) once the Rule Base (RB) has been obtained. The classic approaches to perform genetic tuning [14, 15] consist of using a GA in order to refine the definition parameters that identify the MFs associated to the linguistic terms comprising the initial DB.

Since the real aim of the genetic tuning process is to find the best global configuration of the MFs and not only to find independently specific ones, this optimization problem can become a hard one when the size of the considered system in terms of the number of variables, rules and, particularly, data samples (typically used to guide the search) is big. Moreover, the computing time consumed by these approaches grows with the complexity of the search space.

In order to deal with this complexity, Distributed Genetic Algorithms (DGAs) [20, 21, 22] are found to be excellent optimization algorithms for high dimensional problems. They are able to take advantage of the parallel hardware and software that has become very affordable and broadly available nowadays. Clear examples in this line are multicore processors and linux clusters [23, 24, 25]. This situation makes them perfect to deal with complex search spaces.

In this work, we present a study on the use of the Distributed Genetic Algorithms for the tuning of FRBS from two points of view: solution quality and computing time improvements. To this end, we analyze the application of a specific Gradual Distributed Real-Coded Genetic Algorithm (GDRCGA) to perform an effective genetic tuning of FRBSs [26]. This algorithm employs eight subpopulations in a hypercube topology [27] and makes use of a particular linguistic rule representation model that was proposed in [17] to perform a genetic *lateral tuning of MFs*. This approach is based on the linguistic 2-tuples representation [28] which simplifies the search space by considering only one parameter per MF and, therefore, eases the derivation of optimal models, particularly in complex or high-dimensional problems.

The empirical performance in solution quality and computing time has been assessed by comparing the results of the distributed approach with those obtained from the specialized sequential algorithm, proposed in [17], to perform a lateral tuning of the MFs. Both methods are applied for the modeling of four well-known regression problems. The results show that the distributed approach achieves better results in terms of quality and execution time as the complexity of the problem grows.

The contribution is structured as follows: in the second section DGAs are presented and briefly discussed. In the third section, we present a brief introduction to FRBSs. Next, lateral tuning of FRBSs problem is stated and an efficient sequential specialized algorithm is reviewed. The fifth section describes the DGA used for FRBS tuning. An empirical evaluation of the distributed algorithm is presented in the sixth section. Finally, we close this chapter with some conclusions and final remarks.

2 Distributed Genetic Algorithms

The availability of extremely fast and low cost parallel hardware in the last few years benefits the investigation on new approaches to existing optimization algorithms. The key of these new approaches is achieving gains not only in time, which is somehow inherent to parallel computation, but also gains in quality of the solutions found.

Generally, there are two ways to parallelize GAs. The first way is by means of local parallelization: fitness evaluation of the individuals and, sometimes, the application of the genetic operators are carried out in a parallel way [29, 30]. The second way is by means of global parallelization: complete subpopulations evolve in parallel [31, 32, 33, 34, 35, 36, 27, 37] (distributed approach or DGAs). While the first one is only achieving gains in time, the second one is also able to improve the global performance of the underlying algorithm, subsequently achieving additional gains in the quality of the final solutions. In fact, DGAs [20, 22] are excellent optimization algorithms and have proven to be an interesting approach when trying to cope with large scale problems and when the classic approaches take too much time to give a proper solution.

In this section, our goal is to present an introductory vision of the distributed models. Firstly, we present a taxonomy of the state of the art of DGAs. Finally, the key elements to obtain a well-designed DGA are presented.

2.1 *Taxonomy of Distributed Genetic Algorithms*

Several categorizations of DGAs can be found in literature [20, 21, 22] according to a wide range of criteria. Some of the most used categories when referring to DGAs are:

- **According to the migration policy:**
  - **Isolated:** no migrations between subpopulations. These DGAs are also known as *Partitioned Genetic Algorithms* [31].
  - **Synchronous:** migrations between subpopulations are synchronized, for example, they are carried out at the same time [31, 32].

  - **Asynchronous:** migrations are carried out when some events occur, generally related to the activity of subpopulations [33].

- **According to the connection schema:**
  - **Static schema:** connections between subpopulations are stablished at the start of the execution and they are not modified.
  - **Dynamic schema:** connection topology changes dynamically along the execution of the algorithm. Connection reconfigurations may occur depending on the degree of evolution of the subpopulations.

- **According to the homogeneity:**
  - **Homogeneous:** genetic operators are the same for all subpopulations as well as parameters, fitness function, coding scheme, etc. The vast majority of DGAs proposed in the literature are homogeneous.
  - **Heterogeneous:** subpopulations are all alike [34, 35, 36]. They can differ from the parameters used, genetic operators, coding scheme, etc. One example of these heterogeneous GAs are the *Gradually Distributed Genetic Algorithms* where genetic operators are applied with different intensities [27].

- **According to the granularity:**
  - **Coarse-grained parallelization:** The population is split into small subpopulations that are assigned to different processors. Each subpopulation evolves independently and simultaneously according to a GA. Periodically, a *migration operator* exchanges individuals among subpopulations, which gives them some additional diversity.
  - **Fine-grained parallelization:** The population is split into a big number of small subpopulations. Generally only one subpopulation is assigned to each processor. The selection and crossover operators are applied considering adjacent individuals. For example, each individual chooses its best neighbor for crossover (see Figure 1) and the resulting individual replaces the original one. When a single individual is assigned to each processor, this type of algorithms are known as **Cellular Genetic Algorithms** [37].

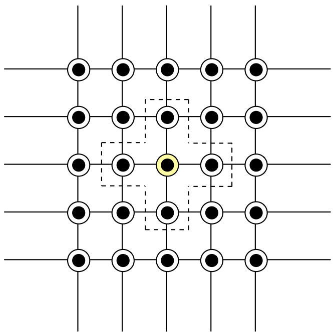


Fig. 1. Cellular Genetic Algorithm: a extreme case of fine-grained parallelization

2.2 Design of Distributed Genetic Algorithms

There are two classic problems [27] in DGAs. A main drawback in DGAs is that the insertion of a new individual coming from a different subpopulation may not be effective. The new individual could be highly incompatible with the receiving subpopulation and therefore it might be ignored or conquer the subpopulation. This probably happens when subpopulations involved are at different stages of evolution.

The arrival of a highly evolved individual comming from a *strong* subpopulation will result in a higher selection ratio than for local individuals which are less evolved. In this way, the subpopulation that sends the highly evolved individual is imposing it to the receiving subpopulation. This problem is known as the *Conquest Problem*.

Symmetrically, when a less evolved individual migrates to a highly evolved subpopulation it will not be selected for reproduction and therefore it will be abandoned. This means a waste of computational and communication efforts. This problem is known as the *Non-effect Problem*.

Both problems could appear in DGAs since subpopulations tend to converge at different speeds. For example, if parameters used for the genetic operators are different, convergence speed will be very different in subpopulations. These problems can directly affect the global convergence leading to non-optimal solutions and losing the efectiveness of the distributed approach.

Subsequently, proposing a well-designed DGA is not a trivial task due to the existence of several factors that can have an influence over the exploration/exploitation balance of the algorithm. There are several elements to consider when designing DGAs:

1. **Topology:** structure of the distributed algorithm which defines relationships between subpopulations and individuals [31, 32, 38, 39]
2. **Migration rate** (MRATE): amount of individuals to be exchanged between subpopulations.
3. **Migration frequency** (MFREQ): number of generations between two consecutive migrations.
4. **Selection strategy:** generally there are two ways of selecting the genetic material to be copied. The first way is randomly selecting an individual from the current subpopulation. The second way consists on selecting the individual with the best fitness in every subpopulation to be copied to another. This last would lead into a more direct evolution because individuals would not have traces of less adapted individuals. The main disadvantage of selecting the best individual is that it could lead into premature convergency [40].
5. **Replacement strategy:** different replacement strategies can be considered, as replacing the worst individuals with the ones received due to migrations, as replacing an individual randomly choosen, etc.
6. **Replication of emigrants:** should individuals be moved, or copied among subpopulations? Exchanging copies of individuals could lead to a highly evolved individuals dominating several less evolved subpopulations [40].

All these parameters have a deep interaction among them and should be carefully determined since a poor choice in one of them can have a strong impact on the global performance of the algorithm. For instance, choosing a extremely high MFREQ can lead to an excessive communication load of the network and the effect of the migrated individuals could be almost imperceptible. Besides, these parameters should be fixed having in mind the hardware that will be used to execute the algorithm: depending on the network it might be better migrating more individuals less frequently than the other way around.

Finally, one procedure for designing DGAs comes from the consideration of spatial separation of subpopulations. Schematically:

```
1. Generate a random population, P.
2. Divide P into $m$ subpopulations: $SP_i$, $i = 1, \dots, m$.
3. Define a topology for $SP_1, \dots, SP_m$.
4. For i = 1 to m do:
      4.1. Apply in parallel during *MFREQ* generations the genetic operators.
      4.2. Send in parallel *MRATE* chromosomes to neighbor subpopulations.
      4.3. Receive in parallel chromosomes from neighbor subpopulations.
5. If stopping criteria is not meet then go back to step 4.
```

3 Fuzzy Rule-Based Systems

FRBSs constitute one of the main contributions of fuzzy logic. The basic concepts which underlie these fuzzy systems are those of linguistic variable and fuzzy IF-THEN rule. A linguistic variable, as its name suggests, is a variable whose values are words rather than numbers, e.g., "small", "young", "very hot" and "quite slow". Fuzzy IF-THEN rules are of the general form: if antecedent(s) then consequent(s), where antecedent and consequent are fuzzy propositions that contain linguistic variables. A fuzzy IF-THEN rule is exemplified by "if the temperature is high then the fan-speed should be high". With the objective of modeling complex and dynamic systems, FRBSs handle fuzzy rules by mimicking human reasoning (much of which is approximate rather than exact), reaching a high level of robustness with respect to variations in the system's parameters, disturbances, etc. The set of fuzzy rules of an FRBS can be derived from subject matter experts or extracted from data through a rule induction process.

In this section, we present a brief overview of the foundations of FRBSs, with the aim of illustrating the way they behave. In particular, in Section 3.1, we introduce the important concepts of fuzzy set and linguistic variable. Finally, in section 3.2, we deal with the basic elements of FRBSs.

3.1 Fuzzy Set and Linguistic Variable

A *fuzzy set* is distinct from a crisp set in that its elements belong to it to a certain degree. The core of a fuzzy set is its MF: a surface or line that defines the relationship between a value in the set's domain and its degree of membership. In particular, according to the original ideal of Zadeh [5], membership of an element x to a fuzzy set A, denoted as $\mu_A(x)$ or simply $A(x)$, can vary from 0 (full non-membership) to 1 (full membership), i.e., it can assume all values in the interval $[0,1]$. Clearly, a fuzzy set is a generalization of the concept of a crisp set whose MF takes values in $\{0,1\}$.

The value of $A(x)$ is the degree of membership of x in A. For example, consider the concept of *high temperature* in an environmental context with temperatures distributed in the interval [0, 50] defined in centigrade degrees. Clearly 0°C is not understood as a “high temperature” value, and we may assign a null value to express its degree of compatibility with the high temperature concept. In other words, the membership degree of 0°C in the class of high temperatures is zero. Likewise, 30°C and over are certainly high temperatures, and we may assign a value of 1 to express a full degree of compatibility with the concept. Therefore, temperature values in the range [30, 50] have a membership value of 1 in the class of high temperatures. From 0°C to 30°C, the degree of membership in the fuzzy set high temperature gradually increases, as exemplified in Figure 2, which actually is a MF $A : T \rightarrow [0,1]$ characterizing the fuzzy set of high temperatures in the universe $T = [0,50]$. In this case, as temperature values increase they become more and more compatible with the idea of high temperature.

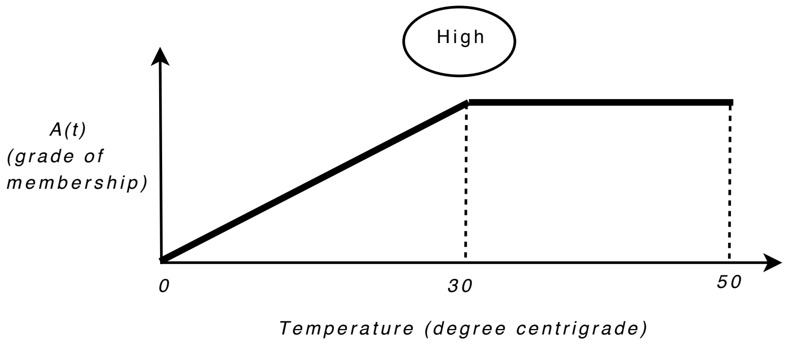


Fig. 2. Membership function

Linguistic variables are variables whose values are not numbers but words or sentences in a natural or artificial language. This concept has clearly been developed as a counterpart to the concept of a numerical variable. In concrete, a linguistic variable L is defined as a quintuple [41]: $L = (x, A, X, g, m)$, where x is the base variable, $A = \{A_1, A_2, \ldots, A_N\}$ is the set of *linguistic terms* of L (called *term-set*), X is the domain (universe of discourse) of the base variable, g is a syntactic rule for generating linguistic terms and m is a semantic rule that assigns to each linguistic term its *meaning* (a fuzzy set in X). Figure 3 shows an example of a linguistic variable *Temperature* with three linguistic terms "Low, Medium, and High". The base variable is the temperature given in appropriate physical units.

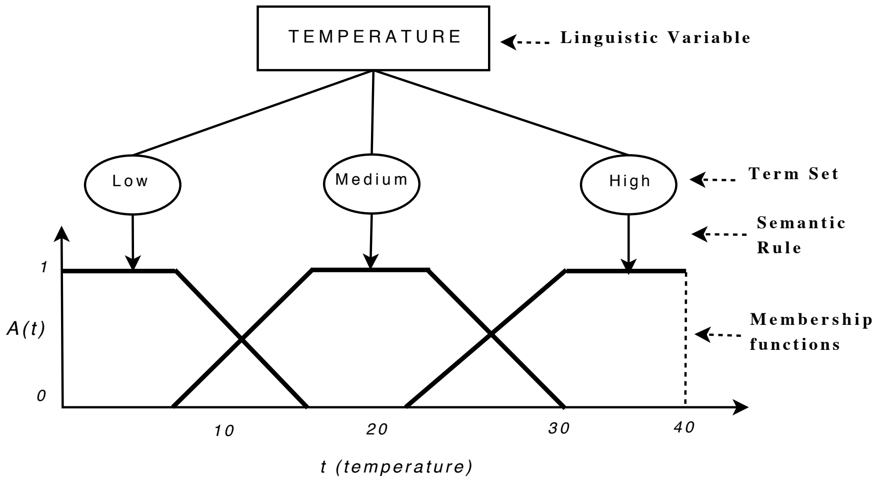


Fig. 3. Example of linguistic variable *Temperature* with three linguistic terms

Each underlying fuzzy set defines a portion of the variable's domain. But this portion is not uniquely defined. Fuzzy sets overlap as a natural consequence of their elastic boundaries. Such an overlap not only implements a realistic and functional semantic mechanism for defining the nature of a variable when it assumes various data values but provides a smooth and coherent transition from one state to another.

3.2 Basic Elements of FRBSs

The essential part of FRBSs is a set of IF-THEN linguistic rules, whose antecedents and consequents are composed of fuzzy statements, related by the dual concepts of fuzzy implication and the compositional rule of inference.

An FRBS is composed of a *knowledge base* (KB), that includes the knowledge in the form of IF-THEN fuzzy rules;

IF *a set of conditions are satisfied*
THEN *a set of consequents can be inferred*

and an inference engine module that includes:

- A *fuzzification interface*, which has the effect of transforming crisp data into fuzzy sets.
- An *inference system*, that uses them together with the KB to make inference by means of a reasoning method.
- A *defuzzification interface*, that translates the fuzzy rule action thus obtained to a real action using a defuzzification method.

FRBSs can be categorized into different families:

- The first includes linguistic models based on collections of IF-THEN rules, whose antecedents are linguistic values, and the system behaviour can be described in natural terms. The consequent is an output action or class to be applied. For example, we can denote them as:
  R_i : If X_{i1} is A_{i1} and $\cdots$ and X_{in} is A_{in} then Y is B_i
  or
  R_i : If X_{i1} is A_{i1} and $\cdots$ and X_{in} is A_{in} then C_k with w_{ik}
  with $i = 1$ to M, and with X_{i1} to X_{in} and Y being the input and output variables for regression respectively, and C_k the output class associated to the rule for classification, with A_{i1} to A_{in} and B_i being the involved antecedents and consequent labels, respectively, and w_{ik} the certain factor associated to the class. They are usually called *linguistic* FRBSs or *Mamdani* FRBSs [42].
- The second category based on a rule structure that has fuzzy antecedent and functional consequent parts. This can be viewed as the expansion of piece-wise linear partition represented as
  R_i : If X_{i1} is A_{i1} and $\cdots$ and X_{in} is A_{in} then $Y = p(X_{i1}, \cdots, X_{in})$,
  with $p(\cdot)$ being a polynomial function, usually a linear expression, $Y = p_0 + p_1 \cdot X_{i1} + \cdots + p_n \cdot X_{in}$. The approach approximates a nonlinear system with a combination of several linear systems. They are called *Takagi and Sugeno's* type fuzzy systems [43] (TS-type fuzzy systems).
- Other kind of fuzzy models are the approximate or scatter partition FRBSs, which differ from the linguistic ones in the direct use of fuzzy variables [44]. Each fuzzy rule thus presents its own semantic, i.e., the variables take different fuzzy sets as values (and not linguistic terms from a global term set). The fuzzy rule structure is then as follow:
  R_i : If X_{i1} is $\hat{A}_{i1}$ and $\cdots$ and X_{in} is $\hat{A}_{in}$ then Y is $\hat{G}_i$
  with $\hat{A}_{ij}$ to $\hat{A}_{in}$ and $\hat{G}_i$ being fuzzy sets. The major difference with respect to the rule structure considered in linguistic FRBSs is that rules of approximate nature are semantics free whereas descriptive rules operate in the context formulated by means of the linguistic semantics.

In linguistic FRBSs, the KB is comprised by two components, a *data base* (DB) and a *rule base* (RB).

- A DB, containing the linguistic term sets used in the linguistic rules and the MFs defining the semantics of the linguistic labels.

Each linguistic variable involved in the problem will have associated a fuzzy partition of its domain represented by the fuzzy sets associated to its linguistic terms. Figure 4 shows an example of fuzzy partition with five labels. In this case, the linguistic term set for each variable (denoted in a common way by *y*) is {Negative Medium (NM), Negative Small (NS), Zero (ZR), Positive Small (PS), Positive Medium (PM)}, which has associated the fuzzy partition of their corresponding domains shown in the Figure. This can be considered as a discretization approach for continuous domains where we establish a membership degree to the items (labels), we have an overlapping between them, and the inference engine manages the matching between the patterns and the rules providing an output according to the rule consequents with a positive matching. The determination of the fuzzy partitions is crucial in fuzzy modelling [45], and the granularity of the fuzzy partition plays an important role in the FRBS behaviour [46].

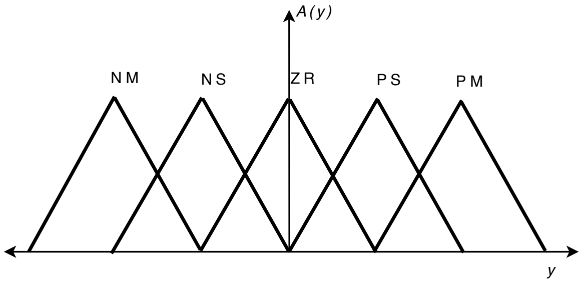


Fig. 4. Membership functions of the linguistic variables (where *y* stands for each variable involved in the system)

If we manage approximate FRBSs, then we do not have a DB due to the fact that rules use fuzzy values rather than linguistic terms.

- A RB comprises of a collection of linguistic rules that are joined by a rule connective (“also” operator). In other words, multiple rules can fire simultaneously for a given input.

The inference engine of an FRBS acts in a different way depending on the kind of problem (classification or regression) and the kind of fuzzy rules (linguistic ones, TS-ones, etc). It usually includes a fuzzification interface that serves as the input to the fuzzy reasoning process; an inference system that infers from the input to several resulting output (fuzzy set, class, etc); and the defuzzification interface or output interface that converts the fuzzy sets obtained from the inference process into a crisp action that constitutes the global output of the FRBS. This last component appears in the case of regression problems, or provide the final class associated to the input pattern according to the inference model.

The generic structure of an FRBS is shown in Figure 5.

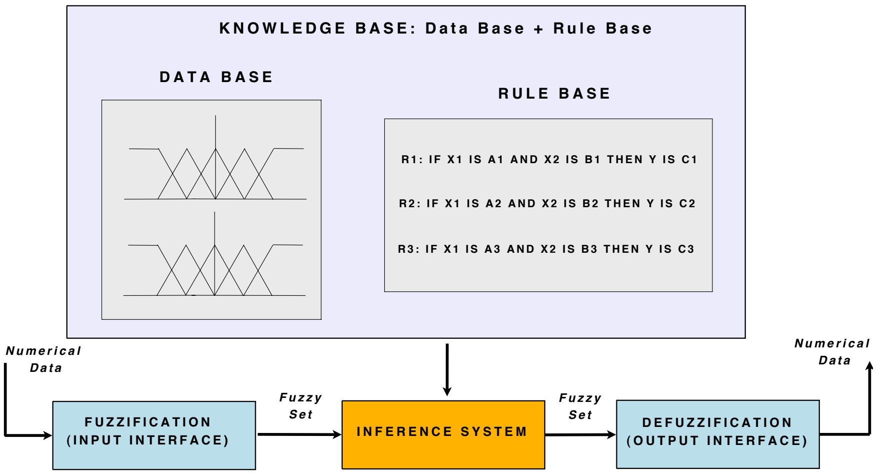

KNOWLEDGE BASE: Data Base + Rule Base
DATA BASE
RULE BASE
R1: IF X1 IS A1 AND X2 IS B1 THEN Y IS C1
R2: IF X1 IS A2 AND X2 IS B2 THEN Y IS C2
R3: IF X1 IS A3 AND X2 IS B3 THEN Y IS C3
Numerical Data
FUZZIFICATION (INPUT INTERFACE)
Fuzzy Set
INFERENCE SYSTEM
Fuzzy Set
DEFUZZIFICATION (OUTPUT INTERFACE)
Numerical Data

Fig. 5. Structure of an FRBS

For more information about fuzzy systems the following books may be consulted [1, 2, 3, 4, 9, 47, 48]. For different issues associated to the trade-off between interpretability and accuracy of FRBSs, the two following edited books present a collection of contributions in the topic [18, 49].

Finally, we must point out that we can find a lot of applications of FRBSs in all areas of engineering, sciences, medicine, etc. At the present it is very easy to find these applications by using the publisher web search tools and by focusing the search on journals of different application areas.

4 Genetic Tuning of FRBSs

With the aim of making a FRBS performs better, some approaches try to improve the preliminary DB definition or the inference engine parameters once the RB has been derived [9, 10, 11]. In order to do so, a tuning process considering the whole KB obtained (the preliminary DB and the derived RB) is used a posteriori to adjust the MFs or the inference engine parameters. A graphical representation of the tuning process is shown in Figure 6.

Among the different possibilities to perform tuning, one of the most widely-used approaches to enhance the performance of FRBSs is the one focused on the DB definition, usually named *tuning of MFs*, or *DB tuning* [17, 14, 15, 19, 50]. In [14], we can find a first and classic proposal on the tuning of MFs. In this case, the tuning methods refine the parameters that identify the MFs associated to the labels

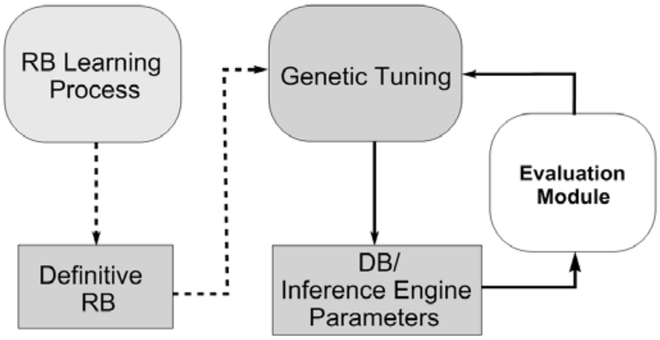


Fig. 6. Genetic tuning process

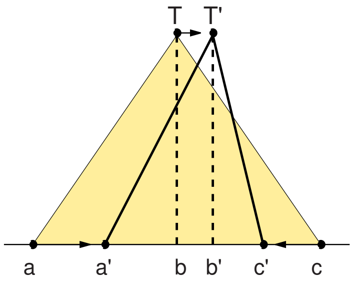


Fig. 7. Tuning by changing the basic MF parameters

comprising the DB. Classically, due the wide use of the triangular-shaped MFs, the tuning methods [19, 9, 15, 14] refine the three definition parameters that identify these kinds of MFs (see Figure 7).

Since the parameters of the MF are interdependent among themselves, in the case of large scale problems, the tuning process becomes an optimization problem on a very complex search space. This, of course, affects the good performance of the optimization methods. A good alternative to solve this problem is the lateral tuning of MFs [17]. This approach makes use of the linguistic 2-tuples representation [28] which simplifies the search space and, therefore, eases the derivation of optimal models, particularly in complex or high-dimensional problems. In order to better handle the complex search space that the tuning of MFs represents, in this work, we analyze the use of the DGAs when performing a lateral tuning of the MFs.

In the next subsection, we describe the efficient lateral tuning of FRBSs. Then, the sequential evolutionary algorithm proposed in [17] to perform the lateral tuning of FRBS is briefly described.

4.1 Lateral Tuning of FRBSs: The Linguistic 2-Tuples Representation

In [17], a new procedure for FRBSs tuning was proposed. It is based on the linguistic 2-tuples representation scheme introduced in [28], which allows the lateral displacement of the support of a label and maintains the interpretability at a good level. This proposal introduces a new model for rule representation based on the concept of symbolic translation [28]. The symbolic translation of a label is a number in $[-0.5, 0.5)$ which expresses its displacement between two adjacent lateral labels (see Figure 8.a). Let us consider a generic linguistic fuzzy partition $S = \{s_0, \ldots, s_{L-1}\}$ (with L representing the number of labels). Formally, we represent the symbolic translation of a label s_i in S by means of the 2-tuple notation,

$$(s_i, \alpha_i), \;\; s_i \in S, \; \alpha_i \in [-0.5, 0.5). \tag{1}$$

The symbolic translation of a label involves the lateral variation of its associated MF. Figure 8 shows the symbolic translation of a label represented by the 2-tuple $(s_2, -0.3)$ together with the associated lateral variation.

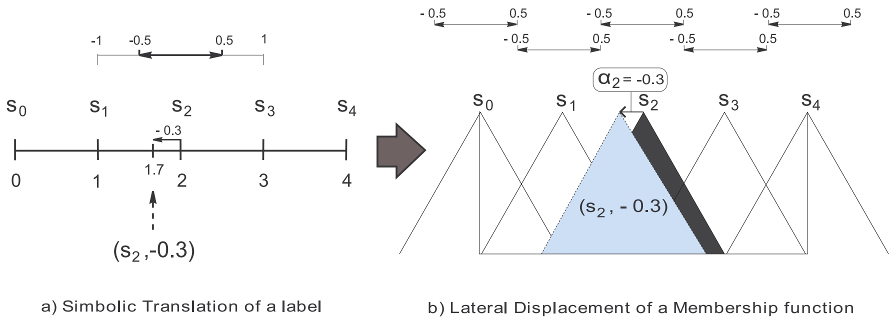

-1
-0.5
0.5
1
s0
s1
s2
s3
s4
- 0.3
0
1
1.7
2
3
4
(s2 ,-0.3)
a) Simbolic Translation of a label
- 0.5
0.5
α2 = -0.3
(s2 , - 0.3)
b) Lateral Displacement of a Membership function

Fig. 8. Symbolic Translation of a Label and Lateral Displacement of the associated MF

In the context of FRBSs, the linguistic 2-tuples could be used to represent the MFs used in the linguistic rules. This way to work, introduces a new model for rule representation that allows the tuning of the MFs by learning their respective lateral displacements. Next, we present this approach by considering a simple control problem.

Let us consider a control problem with two input variables (X_1, X_2), one output variable (Y) and an initial DB defined by experts to determine the MFs for the following labels:

- X_1: $Error \rightarrow \{Negative, Zero, Positive\}$
- X_2: $\nabla Error \rightarrow \{Negative, Zero, Positive\}$
- Y: $Power \rightarrow \{Low, Medium, High\}$

Based on this DB definition, examples of classic and linguistic 2-tuples represented rules are:

- *Classic Rule*:
  R_i: If the **Error** is Zero and the ∇Error is Positive Then the **Power** is High.
- *Rule with 2-Tuples Representation*:
  R_i: If the **Error** is (Zero,0.3) and the ∇Error is (Positive, -0.2) Then the **Power** is (High, -0.1).

With respect to the classic tuning, usually considering three parameters in the case of triangular MFs, this way to work involves a reduction of the search space that eases a fast derivation of optimal models, improving the convergence speed and avoiding the necessity of a large number of evaluations.

In [17], two different rule representation approaches have been proposed, a global approach and a local approach. The global approach tries to obtain more interpretable models, while the local approach tries to obtain more accurate ones. In our case, tuning is applied at the level of linguistic partitions (global approach). By considering this approach, the label s_i^v of a variable v is translated with the same α_i^v value in all the rules where it is used, i.e., a global collection of 2-tuples is used in all the fuzzy rules.

Notice that from the parameters α_i^v applied to each label we could obtain the equivalent triangular MFs. Thus, an FRBS based on linguistic 2-tuples can be represented as a classic Mamdani FRBS [51]. Refer to [17] for further details on this approach.

4.2 Sequential Algorithm for the Lateral Tuning of FRBSs

In [17], an effective sequential GA was proposed to perform a lateral tuning of previously obtained FRBSs. A short description of this algorithm is given below (see [17] for a detailed description).

As the basis optimization procedure the genetic model of CHC [52] was used. Evolutionary model of CHC makes use of a "Population-based Selection" approach. N parents and their corresponding offsprings are combined to select the best N individuals to compose the next population. The CHC approach makes use of an incest prevention mechanism and a restarting process to provoke diversity in the population, instead of the well known mutation operator.

This incest prevention mechanism is considered in order to apply the crossover operator, i.e., two parents are crossed if their hamming distance divided by 2 is higher than a predetermined threshold, T. Since a real coding scheme is considered, each gene is transformed by considering a Gray Code with a fixed number of bits per gene ($BITSGENE$) determined by the system expert. In our case, the threshold value is initialized as:

$$T = (\#GenesC_T * BITSGENE)/4.0. \tag{2}$$

Following the original CHC scheme, T is decreased by one when the population does not change in one generation. In order to avoid very slow convergence, T is also decreased by one when no improvement is achieved with respect to the best chromosome of the previous generation. The algorithm restarts when T is below zero. A scheme of the evolutionary model of CHC is shown in Figure 9.

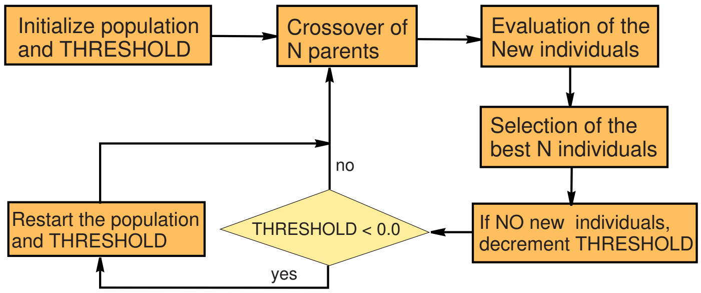

Initialize population and THRESHOLD
Crossover of N parents
Evaluation of the New individuals
Selection of the best N individuals
If NO new individuals, decrement THRESHOLD
THRESHOLD < 0.0
no
yes
Restart the population and THRESHOLD

Fig. 9. Scheme of CHC

In the following, the components used to design the evolutionary tuning process are explained. They are: DB codification and initial gene pool, fitness function, crossover operator and restarting process.

4.2.1 Data Base Codification and Initial Population

A real coding scheme is considered, i.e., the real parameters are the GA representation units (genes). Let us consider n system variables and a fixed number of labels per variable L. Then, a chromosome has the following form (where each gene is associated to the tuning value of the corresponding label),

$$(\alpha_1^1, \ldots, \alpha_1^L, \alpha_2^1, \ldots, \alpha_2^L, \ldots, \alpha_n^1, \ldots, \alpha_n^L) \tag{3}$$

To make use of the available information, the initial FRBS obtained from an automatic fuzzy rule learning method is included in the population as an initial solution. To do so, the initial pool is obtained with the first individual having all genes with value ‘0.0’, and the remaining individuals generated at random in [-0.5, 0.5).

4.2.2 Fitness Function

To evaluate a given chromosome the well-known Mean Square Error (MSE) is used:

$$\text{MSE} = \frac{1}{2 \cdot N} \sum_{l=1}^{N} (F(x^l) - y^l)^2, \tag{4}$$

with N being the data set size, $F(x^l)$ being the output obtained from the FRBS decoded from the said chromosome when the l-th example is considered and y^l being the known desired output.

4.2.3 Crossover Operator

The crossover operator is based on the the concept of environments. These kinds of operators show a good behavior in real coding. Particularly, the BLX-α operator [53] is considered.

This operator allows tuning the degree of exploration and exploitation of the crossover in an easy way. BLX-α crossover works as follows: let us assume that $X = (x_1, \ldots, x_g)$ and $Y = (y_1, \ldots, y_g)$ with $x_i, y_i \in [a_i, b_i) = [-0.5, 0.5) \subset \mathbb{R}(i = 1, \ldots, g)$ are the two real-coded chromosomes that are going to be crossed. Using the BLX-α crossover, one descendant $Z = (z_1, \ldots, z_g)$ is obtained, where z_i is randomly (uniformly) generated within the interval $[l_i, u_i]$, with $l_i = max\{a_i, c_{min} - A\}$, $u_i = min\{b_i, c_{max} + A\}$, $c_{min} = min\{x_i, y_i\}$, $c_{max} = max\{x_i, y_i\}$ and $A = (c_{max} - c_{min}) \cdot \alpha$.

Figure 10 shows how the BLX-α operator works at different stages of the evolution process.

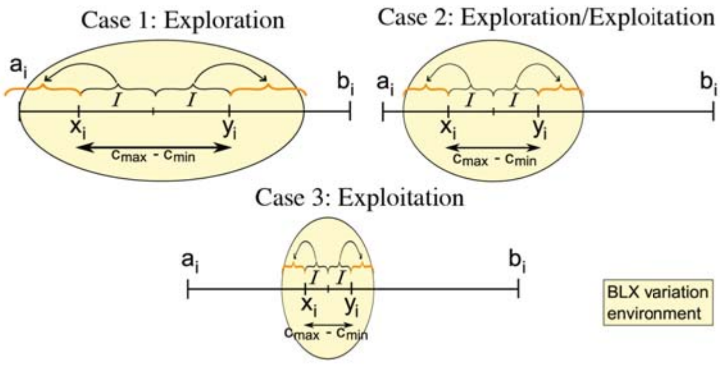


Fig. 10. Different cases/stages of the application of the BLX-α crossover operator, where $\alpha = 0.5$

4.2.4 Restarting Process

To get away from local optima, this algorithm uses a restart approach [52]. In this case, the best chromosome is maintained and the remaining are generated at random within the corresponding variation intervals [-0.5, 0.5). It follows the principles of CHC [52], performing the restart procedure when the threshold T is below zero.

5 A Distributed Genetic Algorithm for the Lateral Tuning of FRBSs

One of the problems when performing tuning with complex data sets is the complexity of the search space. Sometimes even an advanced GA can not deal with the complex search space in terms of time and quality of the results.

Gradual Distributed Real-Coded Genetic Algorithms (GDRCGAs) are a kind of heterogeneous DGAs based on real coding where subpopulations apply genetic operators in different levels of exploitation/exploration. This heterogeneous application of genetic operators produce a *parallel multiresolution* which allows a wide exploration of the search space and effective local precision. Due to appropiate connections between subpopulations in order to gradually exploit multiresolution, these algorithms achieve refinement or expansion of the best emerging zones of the search space.

In order to analyze how DGAs can help the tuning problem, we have selected an efficient GDRCGA [27], that keeps a good balance between exploration and exploitation of the search space. As we said before, we apply this algorithm to perform a lateral tuning of previously obtained FRBSs. In this section, we describe the different characteristics of the DGA used: topology, migrations scheme and the main components of the different subpopulations.

5.1 Main Components of the DGA

The GDRCGA [27] used for FRBS tuning employs 8 subpopulations in a hypercube topology as seen in Figure 11.

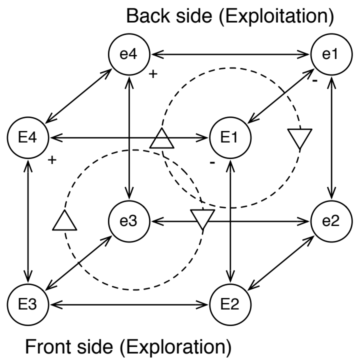


Fig. 11. Hypercube topology for GDRCGA

In this topology two important groups of subpopulations can be clearly identified:

1. **Front side:** this side of the hypercube is oriented to explore the search space. In this side, four subpopulations, $E_1,...,E_4$, apply genetic operators adapted for exploration in a clockwise increasing degree.
2. **Back side:** subpopulations in the back side of the hypercube, $e_1,...,e_4$, apply exploitation oriented genetic operators in a clockwise increasing degree.

One of the key elements of DGAs is the migration policy of individuals between subpopulations. In this particular model, an *immigration* process [54] is achieved since the best chromosome in every subpopulation abandons it and moves to an immediate neighbor. Due to this immigration policy, three different immigration movements can be identified depending on the subpopulations involved:

1. **Refinement migrations:** individuals in the back side move clockwise to the immediate neighbor, i.e. from e_2 to e_3. Chromosomes in the front side move counterclock from a more exploratory subpopulation to a less exploratory oriented one.
2. **Expansion migrations:** individuals in the back side move counterclock to the immediate neighbor and chromosomes in the front side move clockwise from a less exploratory subpopulation to a more exploratory oriented one, i.e. from E_4 to E_1.
3. **Mixed migrations:** subpopulations from one side of the hypercube exchange their best individual with the counterpart subpopulation in the other side: interchange between E_i and e_i, $i = 1 \dots 4$.

Figure 12 shows the three different migration movements described above.

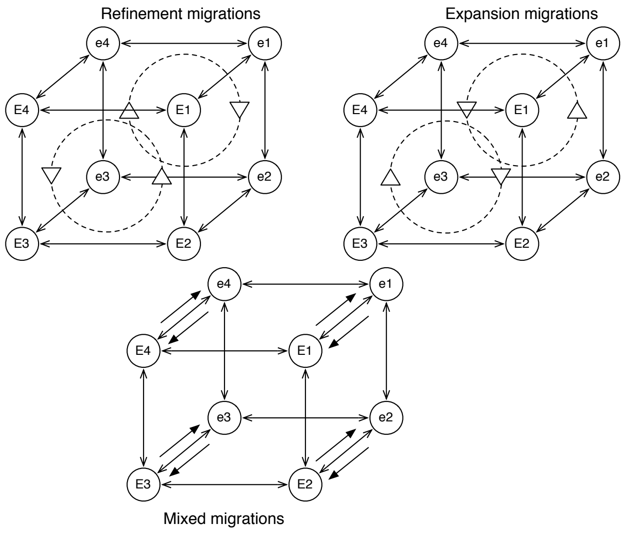


Fig. 12. Three different migration movements for the GDRCGA

As stated in [27], the frequency in which migration movements occur is crucial to avoid the classic withdraws of DGAs: the conquest and noneffect problems. In order to reduce the negative effect of these problems, immigrants stay in the receiving subpopulations for a brief number of generations. Besides, a restart operator is used to avoid stagnation of the search process. This restart operator randomly reinitializes all subpopulations if non-significant improvement of the best element is achieved for a number of generations. Also an elitism strategy is used in order to keep the best adapted individual of every subpopulation.

5.2 Common Components of Individual Subpopulations

The main component used in the different subpopulations of the distributed model are:

- **DB codification and initial subpopulations:** the coding scheme used to represent the displacement parameters is the same one described in section 4.2.1 for the specialized sequential algorithm. Each subpopulation is also initialized in the same way explained in section 4.2.1, i.e., by including the initial FRBS as the first individual in each subpopulation and the remaining individuals generated at random.
- **Selection mechanism:** linear ranking selection (LRS) [55] with stochastic universal sampling [56]. Using LRS the selective pressure can be easily adjusted. In LRS, the individuals are sorted in order of decreasing raw fitness, and then the selection probability, p_s, of each individual I_i is computed according to its rank *rank(I_i)*, with $rank(I_{best}) = 1$, by using the following non-increasing assignment function:
$$p_s(I_i) = \frac{1}{N} \cdot (\eta_{max} - (\eta_{max} - \eta_{min}) \cdot \frac{rank(I_i) - 1}{N - 1}) \qquad (5)$$
where N is the population size, and $\eta_{min} \in [0,1]$ specifies the expected number of copies for the worst individual. The selection pressure is determined by η_{min}. If η_{min} is low, high pressure is achieved. The values of η_{min} used for each subpopulation are shown in Table 1.

Table 1. Values of η_{min} for each subpopulation

Exploitation				Exploration			
$+$	$\leftarrow$		$-$	$-$	$\rightarrow$		$+$
e_4	e_3	e_2	e_1	E_1	E_2	E_3	E_4
0,9	0,7	0,5	0,1	0,9	0,7	0,5	0,1

- **Crossover operator:** the crossover operator used, BLX-α, is the same that was used in the specialized sequential algorithm and it is described in section 4.2.3. As stated before, distinct parameter values are used between subpopulations in order to achieve different degrees of exploitation/exploration. The values used for each subpopulation are shown in Table 2.

  In the absence of selection pressure, values of α, which are $\alpha < 0.5$ make the subpopulations converge towards values in the center of their ranges, producing low diversity levels in the population and inducing a possible premature convergence towards non-optimal solutions. Only when $\alpha = 0.5$, there is a balanced relationship reached between convergence (exploitation) and divergence (exploration). In this case, the probability that a gene will lie in the exploration interval is equal to the probability that it will lie in an exploration interval [53].

Table 2. Values of α for each subpopulation

Exploitation				Exploration			
$+$	$\leftarrow$		$-$	$-$	$\rightarrow$		$+$
e_4	e_3	e_2	e_1	E_1	E_2	E_3	E_4
0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8

- **Mutation operator:** non-uniform mutation operator [57] applied with probability $P_{mut} = 0.125$. This operator is one of the most used mutation operators in real-coded GAs. The mutation operator works as follows. Let $X^t = (x_1^t, \ldots, x_n^t)$ the chromosome selected for mutation. This operator generates a mutated chromosome $X^{t+1} = (x_1^{t+1}, \ldots, x_n^{t+1})$ where:

$$x_i^{t+1} = \begin{cases} x_i^t + \Delta(t, x_i^u - x_i^t) & \text{if } r \leq 0.5 \\ x_i^t - \Delta(t, x_i^t - x_i^l) & \text{otherwise} \end{cases} \tag{6}$$

  where t is the current generation number and r is an uniformly distributed random number between 0 and 1. x_i^l and x_i^u are lower and upper bounds of the i-th gene of the chromosome. The function $\Delta(t,y)$ is defined as follows:

$$\Delta(t,y) = y(1 - u^{(1-\frac{t}{T})^b}) \tag{7}$$

  where u is an uniformly distributed random number between 0 and 1, T is the maximum number of generations and b is a parameter determining the strength of the mutation.

6 Empirical Evaluation

In this section, we analize the empirical results we obtained in order to assess the merits of the distributed approach when applied to lateral tuning of MFs. To evaluate the usefulness of the studied approach, we have used four real-world problems. Table 3 summarizes the main characteristics of the four datasets and shows the link to the KEEL software tool webpage [58] (http://www.keel.es/) from which they can be downloaded.

The studied distributed algorithm described in Section 5 (GDRCGA) is compared with the specialized sequential GA (CHC) [17] in terms of quality of the solutions achieved (MSE) as well as in running time. In both cases, the well-known ad-hoc data-driven learning algorithm of Wang and Mendel [59] is applied to obtain an initial set of candidate linguistic rules. The initial linguistic partitions are comprised of five linguistic terms in the case of datasets with less than 9 variables

Table 3. Data sets used to evaluate the algorithm

Data set	Variables	Instances
Electrical Maintenance	5	1056
Abalone	9	4177
Weather-Izmir	10	1461
Treasury	16	1049

and three linguistic terms in the remaining ones. We consider strong fuzzy partitions of triangular-shaped MFs. Once the initial RB is generated, the different post-processing algorithms can be applied.

A five-fold cross-validation approach has been used, i.e., we randomly split the data set into 5 folds, each containing the 20% of the patterns of the data set, and used four folds for training and the other for testing. So, a total of five runs have been carried out with different independent test sets. For each dataset, we therefore consider the average results of the five runs. The average results of the initial FRBSs obtained by the Wang and Mendel algorithm are shown in Table 4 (initial reference results).

Table 4. Initial mean squared errors and deviations in both, training and test sets, obtained by Wang & Mendel

Dataset	Training	σ_{tra}	Test	σ_{test}
Electrical M.	57606	2841	57934	4733
Treasury	1.636	0.121	1.632	0.182
Weather-Izmir	6.944	0.720	7.368	0.909
Abalone	3.341	0.130	3.474	0.247

An interesting point to be taken into account is the *evolution* of the MSE as the number of evaluations increases. So, three different numbers of evaluations have been choosen: 10000, 25000 and 50000 evaluations per run. The results in terms of quality of the solutions attained are shown in Table 5. An important fact to notice is that the MSE achieved with the distributed method is lower than the error obtained with the specialized GA in all data sets at 50000 evaluations. The distribuited approach also obtains good results with fewer iterations in some cases (e.g. Electrical Maintenance with 25000 iterations), but clearly its real effectiveness will be reached when the computation load is higher.

Generally, when comparing a distributed or parallel approach with any other sequential algorithm, an interesting measure is the execution time gain ratio. This ratio could be defined as follows:

$$R = \frac{T_{seq}}{T_{dist}} \tag{8}$$

Table 5. Mean squared errors in training and test sets obtained by CHC and GDRCGA. The winner for each pair of training is *in italics*. The winner for each pair of test is **boldfaced**

		CHC		GDRCGA	
Data set	Evaluations	Training	Test	Training	Test
Electrical	10000	*2.59363671E+04*	2.92591821E+04	2.65539710E+04	**2.89024830E+04**
Maintenance	25000	2.48690100E+04	2.80510895E+04	*2.39248797E+04*	**2.67720415E+04**
	50000	2.46214328E+04	2.78282761E+04	*2.26682075E+04*	**2.54097540E+04**
	10000	*2.61355003E+00*	2.79981355E+00	2.65916770E+00	**2.79026550E+00**
Abalone	25000	2.60333453E+00	2.79298130E+00	*2.59992700E+00*	**2.76143590E+00**
	50000	2.60303744E+00	2.79117626E+00	*2.57035010E+00*	**2.75904570E+00**
	10000	*1.68875432E+00*	**1.89318352E+00**	1.89195950E+00	1.95458830E+00
Weather-Izmir	25000	*1.64117336E+00*	**1.86996710E+00**	1.66238700E+00	1.87669540E+00
	50000	1.64010963E+00	1.86891124E+00	*1.57019250E+00*	**1.86195430E+00**
	10000	*1.71238672E-01*	**1.86722425E-01**	2.12486800E-01	2.16882700E-01
Treasury	25000	*1.33618274E-01*	**1.50895419E-01**	1.42194200E-01	1.67407400E-01
	50000	1.20604483E-01	1.37784224E-01	*1.15845500E-01*	**1.31803000E-01**

where T_{seq} is the time spent by the sequential algorithm and T_{dist} is the execution time of the distributed approach. The higher the value of R, the better. Time gain ratio values obtained in the empirical experimentation are shown in Table 6.

Table 6. Time gain ratio with 50000 evaluations

Data set	T_{seq}	T_{dist}	R
Electrical Maintenance	187,3	391,6	0,479
Trasury	525,3	739,7	0,710
Weather-Izmir	849,8	867,1	0,980
Abalone	1980,9	942,5	2,101

In Table 6, the time gain ratio, R, increases with the problem complexity. In the less complex data sets the ratio obtained is substantially lower because the sequential specialized GA is very fast and the time spent in communications of the distributed approach slows it down in comparison. As the complexity of the data set increases the time gain ratio also increases, showing that the distributed approach in the most complex data set is more than two times faster than the sequential specialized GA. The distributed algorithm takes longer than the sequential algorithm when dealing with small size data sets mainly due to two reasons: interprocess communication in the distributed approach implies additional execution time which can not be parallelized and the specialized algorithm is optimized for small size data sets where the search space is not too complex.

On the other hand, observation of the evolution of the MSE is also an interesting factor to take into account. Two different data sets have been choosen in order to study the evolution of the MSE: Electrical Maintenance and Treasury. These two data sets were choosen because of their different complexity: Treasury data set is far more complex than Electrical Maintenance.

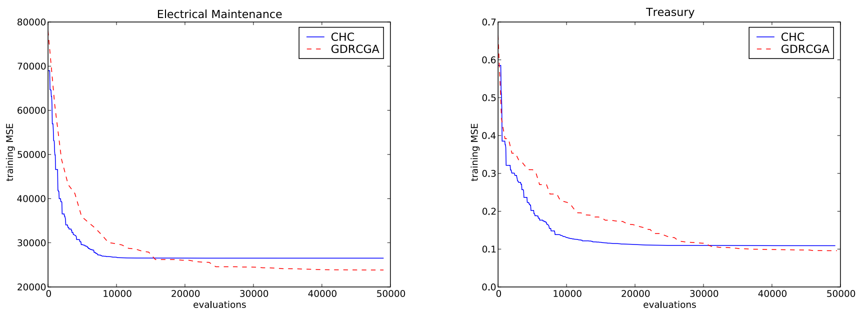
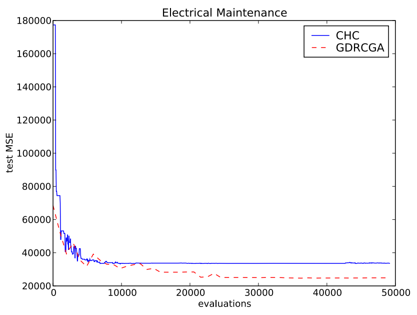


Fig. 13. Evolution of the MSE in training (convergence): Electrical Maintenance and Treasury datasets

Figure 13 shows the convergence of both algorithms in both problems. Due to the distributed nature of the algorithm and consequently the spatial separation implied, it needs more evaluations to converge than the sequential algorithm. It always presents the same behaviour in comparision to the sequential approach: with a small number of evaluations it yields a higher error than the sequential one, but when the number of evaluations is high it gives solutions with a better quality.

As it has been stated, the distributed approach needs more iterations to achieve convergency for complex data sets. This situation can be observed in Figure 13 (right side): GDRCGA achieves better MSE values when the search process has consumed two thirds of the number of evaluations. On the other hand, when dealing with less complex data sets like Electrical Maintenance (Figure 13 left side), the distributed approach quickly achieves better MSE values from almost the begining of the search process and keeps gaining distance from the sequential CHC algorithm. In fact, GDRCGA begins achieving better MSE values shortly after the search process has consumed one third of the number of evaluations available. These two situations can be also verified in Table 5.

Besides studying the evolution of the MSE in training (convergence), it is also interesting to analyze the effects that it produces on the MSE in test regarding the same data sets. Figure 14 shows the MSE in test of Treasury and Electrical Maintenance datasets. Again, we can observe that the distributed approach needs more evaluations to outperform the sequential algorithm in the more complex problem (see Figure 14 right side) while better results are obtained practically from the beginning in the simpler one (see Figure 14 left side). However, two interesting characteristics can be highlighted. Firstly, the evolution in the test error shown by GDRCGA seems quite more stable in both problems. Secondly, GDRCGA shows practically the same trend in training and test in both datasets, while the sequential approach worsen the test error once the half of the evaluations are consumed in the more complex dataset (overfitting). These characteristic are quite recommendable in the fuzzy modeling framework.



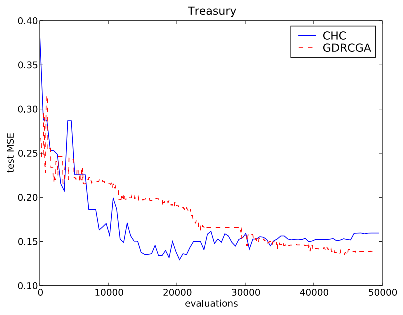


Fig. 14. Effects on the MSE in test: Electrical Maintenance and Treasury data sets

7 Conclusions and Final Remarks

In this chapter, we have presented an study on the use of the DGAs for the lateral tuning of FRBSs. To this end, we have analyzed the performance of a specific GDR-CGA employing 8 subpopulations in a hypercube topology [26]. This algorithm has been compared with the specialized GA presented in [17] to perform the lateral tuning of FRBSs.

From the empirical results obtained, we can conclude that as the complexity of the problem grows, the distributed approach outperforms the specialized sequential algorithm. Moreover, the distributed procedure makes effective use of the wall time in relation to the computing times required by the sequential algorithm. Also, when dealing with complex search spaces, the distributed approach is able to converge to better quality solutions than the sequential algorithm. This behaviour makes the distributed tuning algorithm very useful when dealing with large scale problems where the complexity of the search space is high.

Since execution time and quality of the results are two properties always in conflict somehow, the distributed approach could be graduated in order to achieve faster execution times with a small cost in quality and viceversa.

References

1. Driankow, D., Hellendoorn, H., Reinfrank, M.: An introduction to fuzzy control. Springer, Berlin (1993)
2. Pedrycz, W.: Fuzzy Modelling: Paradigms and practice. Kluwer Academic Publishers, Dordrecht (1996)
3. Palm, R., Driankov, D., Hellendoorn: Model based fuzzy control. Springer, Heidelberg (1997)
4. Ishibuchi, H., Nakashima, T., Nii, M.: Classification and modeling with linguistic information granules: Advances approaches to linguistic data mining. Springer, Heidelberg (2004)
5. Zadeh, L.A.: Fuzzy sets. Information and Control 8, 338–353 (1965)

6. Zadeh, L.A.: Outline of a new approach to the analysis of complex systems and decision processes. IEEE Trans. Syst., Man, Cybern. 3, 28–44 (1973)
7. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, New York (1989)
8. Holland, J.H.: Adaptation in natural and artificial systems. The University of Michigan Press, Michigan (1975); The MIT Press, London (1992)
9. Cordón, O., Herrera, F., Hoffmann, F., Magdalena, L.: Genetic Fuzzy Systems: evolutionary tuning and learning of fuzzy knowledge bases. World Scientific, Singapore (2001)
10. Herrera, F.: Genetic fuzzy systems: Taxonomy, current research trends and prospects. Evolutionary Intelligence 1, 27–46 (2008)
11. Cordón, O., Gomide, F., Herrera, F., Hoffmann, F., Magdalena, L.: Ten years of genetic fuzzy systems: current work and new trends. Fuzzy Sets and Systems 141(1), 5–31 (2004)
12. Eiben, A.E., Smith, J.E.: Introduction to evolutionary computation. Springer, Berlin (2003)
13. Zadeh, L.A.: The concept of a linguistic variable and its applications to approximate reasoning, parts i, ii and iii. Information Science 8, 8, 9, 199–249, 301–357, 43–80 (1975)
14. Karr, C.: Genetic algorithms for fuzzy controllers. AI Expert 6(2), 26–33 (1991)
15. Herrera, F., Lozano, M., Verdegay, J.L.: Tuning fuzzy logic controllers by genetic algorithms. International Journal of Approximate Reasoning 12, 299–315 (1995)
16. Alcalá, R., Alcalá-Fdez, J., Casillas, J., Cordón, O., Herrera, F.: Hybrid learning models to get the interpretability-accuracy trade-off in fuzzy modeling. Soft Computing 10(9), 717–734 (2006)
17. Alcalá, R., Alcalá-Fdez, J., Herrera, F.: A proposal for the genetic lateral tuning of linguistic fuzzy systems and its interaction with rule selection. IEEE Transactions on Fuzzy Systems 15(4), 616–635 (2007)
18. Casillas, J., Cordón, O., del Jesus, M.J., Herrera, F.: Accuracy improvements in linguistic fuzzy modeling. Springer, Heidelberg (2003)
19. Casillas, J., Cordón, O., del Jesus, M.J., Herrera, F.: Genetic tuning of fuzzy rule deep structures preserving interpretability and its interaction with fuzzy rule set reduction. IEEE Trans. Fuzzy Syst. 13(1), 13–29 (2005)
20. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell (2000)
21. Fernández de Vega, F., Cantu-Paz, E.: Special issue on distributed bioinspired algorithms. Soft Computing 12(12), 1143–1144 (2008)
22. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. Wiley, Chichester (2005)
23. Sterling, T., Becker, D.J., Savarese, D.F.: How to build a beowulf: A guide to the implementation and application of PC clusters. The MIT Press, Cambridge (1999)
24. Spector, D.H.M.: Building Linux Clusters. O'Reilly, Sebastopol (2000)
25. Dowd, K., Severance, C.: High Performance Computing. O'Reilly, Sebastopol (1998)
26. Robles, I., Alcalá, R., Benítez, J.M., Herrera, F.: Distributed genetic tuning of fuzzy rule-based systems. In: Proceedings of the International Fuzzy Systems Association - European Society for Fuzzy Logic and Technology (IFSA-EUSFLAT) Congress (in press, 2009)
27. Herrera, F., Lozano, M.: Gradual distributed real-coded genetic algorithms. IEEE Transactions on Evolutionary Computation 4(1), 43–63 (2000)
28. Herrera, F., Martínez, L.: A 2-tuple fuzzy linguistic representation model for computing with words. IEEE Trans. Fuzzy Syst. 8(6), 746–752 (2000)

29. Bäck, T., Beielstein, T.: User's group meeting. In: Proceedings of the EuroPVM 1995: Second European PVM, pp. 277–282 (1995)
30. Punch, W., Goodman, E., Pei, M., Chai-shun, L., Hovland, P., Enbody, R.: Further research on feature selection and classification using genetic algorithms. In: Forrest, S. (ed.) Proceedings of the Fifth International Conference on Genetic Algorithms, pp. 557–564 (1993)
31. Tanase, R.: Distributed genetic algorithms. In: Proceedings of the Third International Conference on Genetic Algorithms, pp. 434–439 (1989)
32. Mühlenbein, H., Schomisch, M., Born, J.: The parallel genetic algorithm as function optimizer. Parallel Computing 17(6), 619–632 (1991)
33. Lin, S.C., Punch III, W.F., Goodman, E.D.: Coarse-grain parallel genetic algorithms: Categorization and new approach. In: Proceedings of the Sixth IEEE Parallel and Distributed Processing, pp. 28–37 (1994)
34. Alba, E., Luna, F., Nebro, A., Troya, J.M.: Parallel heterogeneous genetic algorithms for continuous optimization. Parallel Computing 30(5), 699–719 (2004)
35. Schlierkamp-Voosen, D., Mühlenbein, H.: Strategy adaptation by competing subpopulations. In: Davidor, Y., Männer, R., Schwefel, H.-P. (eds.) PPSN 1994. LNCS, vol. 866, pp. 199–208. Springer, Heidelberg (1994)
36. Schnecke, V., Vornberger, O.: An adaptative parallel algorithm for vlsi-layout optimization. In: Ebeling, W., Rechenberg, I., Voigt, H.-M., Schwefel, H.-P. (eds.) PPSN 1996. LNCS, vol. 1141, pp. 22–27. Springer, Heidelberg (1996)
37. Alba, E., Dorronsoro, B.: Cellular Genetic Algorithms. Springer, Heidelberg (2008)
38. Tanase, R.: Parallel genetic algorithm for a hypercube. In: Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications, pp. 177–183 (1987)
39. Cohoon, J.P., Hedge, S., Martin, W.: Punctuated equilibria: A parallel genetic algorithm. In: Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications, pp. 148–154 (1987)
40. Ryan, C.: Niche and species formation in genetic algorithms. In: Chambers, L. (ed.) Practical Handbook of Genetic Algorithms: Applications, pp. 57–74. CRC Press, Boca Raton (1995)
41. Klir, G., Yuan, B.: Fuzzy sets and fuzzy logic; theory and applications. Prentice-Hall, Englewood Cliffs (1995)
42. Mamdani, E.H.: Application of fuzzy algorithms for control of simple dynamic plant. Proc. Inst. Elect. Eng. 121(12), 1585–1588 (1974)
43. Takagi, T., Sugeno, M.: Fuzzy identification of systems and its application to modelling and control. IEEE Trans. Syst. Man and Cybernetics 15(1), 116–132 (1985)
44. Alcalá, R., Casillas, J., Cordón, O., Herrera, F.: Building fuzzy graphs: features and taxonomy of learning non-grid-oriented fuzzy rule-based systems. International Journal of Intelligent Fuzzy Systems 11, 99–119 (2001)
45. Au, W.-H., Chan, K., Wong, A.K.C.: A fuzzy approach to partitioning continous attributes for classification. IEEE Transactions on Knowledge and Data Engineering 18(5), 715–719 (2006)
46. Cordón, O., Herrera, F., Villar, P.: Analysis and guidelines to obtain a good fuzzy partition granularity for fuzzy rule-based systems using simulated annealing. International Journal of Approximate Reasoning 25(3), 187–215 (2000)
47. Yager, R., Filev, D.: Essentials of fuzzy modeling and control. John Wiley and Sons, Chichester (1994)
48. Kuncheva, L.: Fuzzy classifier design. Springer, Heidelberg (2000)
49. Casillas, J., Cordón, O., Herrera, F., Magdalena, L.: Interpretability issues in fuzzy modeling. Springer, Heidelberg (2003)

50. Gürocak, H.B.: A genetic-algorithm-based method for tuning fuzzy logic controllers. Fuzzy Sets and Systems 108(1), 39–47 (1999)
51. Mamdani, E.H., Assilian, S.: An experiment in linguistic synthesis with a fuzzy logic controller. International Journal of Man-Machine Studies 7, 1–13 (1975)
52. Eshelman, L.J.: The CHC adaptive search algorithm: How to have safe serach when engaging in nontraditional genetic recombination. In: Rawlin, G.J.E. (ed.) Foundations of genetic Algorithms, vol. 1, pp. 265–283. Morgan Kaufman, San Francisco (1991)
53. Eshelman, L.J., Schaffer, J.D.: Real-coded genetic algorithms and interval-schemata. Foundations of Genetic Algorithms 2, 187–202 (1993)
54. Kröger, B., Schwenderling, P., Vornberger, O.: Parallel genetic packing on transputers. In: Parallel Genetic Algorithms: Theory and Applications: Theory Applications, pp. 151–186 (1993)
55. Baker, J.E.: Adaptive selection methods for genetic algorithms. In: Proceedings of the First International Conference on Genetic Algorithms and their Applications, pp. 101–111. Erlbraum Associates, Hillsdale (1985)
56. Baker, J.E.: Reducing bias and inefficiency in the selection algorithm. In: Proceedings of the 2nd International Conference on Genetic Algorithms, ICGA 1987 (1987)
57. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer, Heidelberg (1992)
58. Alcalá-Fdez, J., Sánchez, L., García, S., del Jesus, M.J., Ventura, S., Garrell, J.M., Otero, J., Romero, C., Bacardit, J., Rivas, V.M., Fernández, J.C., Herrera, F.: KEEL: A software tool to assess evolutionary algorithms to data mining problems. Soft Computing 13(3), 307–318 (2009)
59. Wang, L.X., Mendel, J.M.: Generating fuzzy rules by learning from examples. IEEE Trans. Syst. Man and Cybernetics 22(6) (1992)

Parallel and Distributed Optimization of Dynamic Data Structures for Multimedia Embedded Systems

José L. Risco-Martín, David Atienza, J. Ignacio Hidalgo, and Juan Lanchares

Abstract. Energy-efficient design of multimedia embedded systems demands optimizations in both hardware and software. Software optimization has no received much attention, although modern multimedia applications exhibit high resource utilization. In order to efficiently run this kind of applications in embedded systems, the dynamic memory subsystem needs to be optimized. A key role in this optimization is played by the Dynamic Data Types (DDTs) that reside in every real-life application. It would be desirable to organize this set of DDTs to achieve the best performance in the target embedded system. This problem is NP-complete, and cannot be fully explored. In these cases the use of parallel processing can be very useful because it allows not only to explore more solutions spending the same time, but also to implement new algorithms. In this work, we propose a method that uses parallel processing and evolutionary computation to explore DDTs in the design of embedded applications. We propose a parallel *Multi-Objective Evolutionary Algorithm (MOEA)* which combines NSGA-II and SPEA2. We use *Discrete Event Systems Specification (DEVS)* to implement this parallel evolutionary algorithm over *Service Oriented Architecture (SOA)*. Parallelism improves the solutions found by the corresponding sequential algorithms, and it allows system designers to reach better solutions than previous approximations.

José L. Risco-Martín · J. Ignacio Hidalgo · Juan Lanchares
Dept. of Computer Architecture and Automation (DACYA), Complutense University of Madrid (UCM), C/Prof. José García Santesmases, s/n., 28040 Madrid, Spain
e-mail: {jlrisco,hidalgo,julandan}@dacya.ucm.es

David Atienza
Dept. of Computer Architecture and Automation (DACYA), Complutense University of Madrid (UCM), C/Prof. José García Santesmases, s/n., 28040 Madrid, Spain
e-mail: datienza@dacya.ucm.es
and
Embedded Systems Laboratory (ESL), Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

1 Introduction

Latest multimedia embedded systems typically require reliable and powerful computing, superior graphical performance, multiple I/O configurations and long product life support. Currently, these systems are able to run applications initially designed for high performance desktop computers [6], which having large run-time memory management requirements, need to be mapped onto an extremely compact device. However, embedded systems struggle to execute these complex applications because they hold very different constraints regarding memory usage features. Thus, designers must pay attention in the porting process to the lack of memory of the target embedded system as well as the fact that such systems extensively employ the dynamic memory subsystem.

A desktop application is typically implemented using data structures or *Dynamic Data Types (DDTs)* [2] (dynamic arrays, linked lists, etc) to store their data. The DDT for each container is usually selected to achieve the best performance without bearing in mind other requirements such as power consumption, memory accesses and memory usage, an important factor in embedded systems. Thus, to map a desktop application, designers must reach the best set of DDTs that minimizes the system behavior according to some constraint of the target device, such as memory accesses, memory usage and energy consumption [3]. This is an NP-complete problem and cannot be fully explored.

This task has been typically performed in the past using a pseudo-exhaustive evaluation of the design space of DDTs, including multiple executions of the application, to attain a *Pareto Front (PF)* of solutions [8], which tries to cover all the optimal implementation points for the required design metrics. The construction of this PF has been proven a very time-consuming process, sometimes even unaffordable [10]. To solve this problem we propose to use parallel processing based on Multi Objective Evolutionary Algorithms. This combination is very useful because we can explore more solutions in less time. Besides, the use of parallel processing allows us to design new algorithms that improve the number and the quality of solutions. However, before explaining deeply our proposal we begin by reviewing some related work.

Several works have been made in the field of embedded memory subsystem optimization, both in static and dynamic memory. In the case of static memory, Benini et al. [4] and Panda et al. [24] presented in the last decade two thorough surveys on static data and memory optimization techniques for embedded systems. More recently, in [6], [10] and [7], authors achieve to reduce the memory subsystems requirements by 50% using a linear time algorithm by exploring a coordinated data and computation reordering for array-based data structures in multimedia applications. Nevertheless, they are not suitable for exploration of complex DDTs employed in modern multimedia applications.

In the field of dynamic embedded software, there are some approaches that propose power-aware transformation and use pruning strategies based on heuristics to find the best solution [33] [24] [23]. These proposals have some weakness. On the one hand, they need to study and develop efficient pruning cost-function and a fully manual optimization, which is not very efficient. On the other hand, these works do not bear in mind that the final behavior of the system presents inter-dependencies in the set of DDT implementations.

Regarding exploration methods, *Multi-Objective Evolutionary Algorithms (MOEAs)* started to be used to solve different CAD optimization problems (as proposed Michalewicz in [20]). In our case, the use of MOEAs to explore DDTs has become a good alternative. In [6] and [33] several transformations are established for DDTs and static data profiling and static memory access patterns to physical memories, and they are used to obtain information in order to find the best solution. In this context, MOEA-based optimization has been applied to solve linear-and non-linear problems by exploring the entire state space in parallel. Thus, it is possible to perform optimization in non convex regular functions, and to select the order of algorithmic transformations in concrete types of source codes [24]. However, such techniques are not applicable to DDT implementations due to it is not possible to know the DDT behavior (the number of elements stored in the DDT, number of read accesses, number of write accesses, etc.), at compile-time.

In the field of dynamic memory optimizations in embedded systems, Atienza et al. [3] have performed an initial analysis of one single type of MOEA showing the potential benefits of MOEAs for this kind of problems. Nevertheless, their work does not provide a complete analysis of tradeoffs between different technologies of sequential and parallel MOEAs. We tackle this problem in the present research work.

In order to be able to use parallel evolutionary algorithms for multi-objective problems, different paradigms of the parallel processing and their corresponding parameters have to be analyzed. In [31], Veldhuizen studies some important questions in the formulation of *parallel Multi-Objective Evolutionary Algorithms (pMOEA)* such as migration, replacement and niching schemes. Besides, he gives a classification of pMOEA based on the island paradigm: (1) islands execute the same MOEA [34]; (2) islands execute different MOEA [14]; (3) each island evaluates a different subset of objective functions [32]; and (4) each island considers a different region of the search domain [30].

In this work, we propose a new method that uses parallel processing and evolutionary algorithm to explore the design space of DDT implementation. To this end, we use Discrete Event Systems Specifications (DEVS) [35] over Service Oriented Architecture (SOA) [21], which offers DEVS-based simulations as a web service based on standard technologies, called DEVS/SOA. We explore several classical Multi-Objective Evolutionary Algorithms (MOEA) [11] and propose an algorithm which combines NSGA-II and SPEA2 within a DEVS/SOA framework. It allows designers to reach a larger number of solutions than classical approaches. Our parallel design may be included in the second group mentioned before (islands executing different MOEAs). Since our migration policy is synchronous, we have combined two elitist evolutionary algorithms with different complexity, namely *Strength Pareto Evolutionary Algorithm 2 (SPEA2)* [36] and *Non-dominated Sorting Genetic Algorithm II (NSGA-II)* [12], implementing three variations of a pMOEA. SPEA2 is

$O(N^3)$ and NSGA-II is $O(mN^2)$, where N is the population size and m is the number of objectives.

Our experiments in a real-life dynamic embedded application show that: (1) NSGA-II and SPEA2 reach important speed-ups (up to 469× faster) with respect to other traditional heuristics; (2) the parallel algorithm can achieve significant speed-ups with respect to the sequential versions in a multi-core architecture. Moreover, we compare the sequential and parallel approaches by means of multiple metrics, showing that the quality of the solutions is improved by the combination of NSGA-II and SPEA2 in a parallel implementation; and (3) such combination is executed on 16 workstations of two cores each, where several population sizes were deployed as per our experiments. The experiments returned very promising results. In particular, we got empirical evidence that on increasing the size of the population, the performance of the pMOEA improves as we increase the number of workstations used.

The rest of the paper is organized as follows. Definitions of MOEAs and underlying technologies such as DEVS and DEVS/SOA are given in Section 2. In Section 3 the Dynamic Data Types optimization problem is explained. In Section 4, we present our multi-objective optimization process. A description of the MOEAs, including an explanation of our parallel proposal, which combines NSGA-II and SPEA2 algorithms, is also detailed. Section 5 details our experimental setup as well as shows some performance and quality metrics used in our experiments in Section 6. Finally, in Section 7 we summarize the main conclusions and future work.

2 Background

2.1 Multi-objective Evolutionary Algorithms

Multi-objective optimization aims at simultaneously optimizing several objectives sometimes contradictory (memory accesses, memory usage and energy consumption for our problem). For such kind of problems, there does not exist a single optimal solution, and some trade-offs need to be considered. Without any loss of generality, we can assume the following m-objective minimization problem:

$$\begin{array}{ll} Minimize\ \mathbf{z} = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots f_m(\mathbf{x})) & \\ subject\ to \qquad \mathbf{x} \in X & (1) \end{array}$$

where $\mathbf{z}$ is the objective vector with m objectives to be minimized, $\mathbf{x}$ is the decision vector, and X is the feasible region in the decision space. A solution $\mathbf{x} \in X$ is said to dominate another solution $\mathbf{y} \in X$ (denoted as $\mathbf{x} \prec \mathbf{y}$) if the following two conditions are satisfied.

$$\begin{array}{ll} \forall i \in \{1,2,\dots,m\}, f_i(\mathbf{x}) \leq f_i(\mathbf{y}) & \\ \exists i \in \{1,2,\dots,m\}, f_i(\mathbf{x}) < f_i(\mathbf{y}) & (2) \end{array}$$

If there is no solution which dominates $\mathbf{x} \in X$, $\mathbf{x}$ is said to be a *Pareto Optimal Solution (POS)*. The set of all elements of the search space that are not dominated by any other element is called the *Pareto Optimal Front (POF)* of the multi-objective problem: it represents the best possible solution with respect to the contradictory objectives.

In both algorithms, the sequential and parallel versions, we attempt to reach the higher number of non-dominated solutions as possible.

Nowadays, many MOEAs have been developed. They can be classified into two broad categories: non-elitist and elitist, also called first and second generation MOEAs [8]. In the elitist approach, EAs store the best solutions of each generation in an external set. This set will then be a part of the next generation. Thus, the best individuals in each generation are always preserved, and this helps the algorithm to get close to its POF. Algorithms such as PESA-II [9], MOMGA-II [38], NSGA-II and SPEA2 are examples of this category. In contrast, the non-elitist approach does not guarantee preserving the set of best individuals for the next generation [8]. Examples of this category include MOGA [15], HLGA [16], NPGA [18] and VEGA [28].

When implementing a MOEA, the designer has to overcome two major problems [37]. The first problem is how to get close to the POF [11]. The second problem is how to keep diversity among the solutions in the obtained set. These two problems become common criteria for most current algorithmic performance comparisons and they will be used in the experimental results section.

Table 1. Common evolutionary algorithm framework

1. Initialize the Population **P**
2. (elitist EAs) Select elitist solutions from **P** to create external set **EP**
3. Create mating pool from one or both **P** and **EP**
4. Reproduction based on the pool to create the next generation **P** using evolutionary operators
5. (elitist EAs) Combine **EP** into **P**
6. Go to step 2 if the terminated condition is not satisfied

Although all the cited MOEAs are different from each other, we can find some common steps in these algorithms, which are summarized in Table 1. As we have already mentioned, two representative elitist algorithms, namely, SPEA2 and NSGA-II were selected.

2.2 DEVS and DEVSJAVA

DEVS formalism consists of models, the simulator and the experimental frame. We will focus our attention to the specified two types of models i.e. atomic and coupled models. The atomic model is the irreducible model definition that specifies the behavior for any modeled entity. The coupled model is the aggregation/composition of two or more atomic and coupled models connected by explicit couplings. The formal definition of parallel DEVS (P-DEVS) is given in [35]. An atomic model is defined by the following equation:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda \rangle \tag{3}$$

where,

- X is the set of input values
- S is the state space
- Y is the set of output values
- $\delta_{int} : S \rightarrow S$ is the internal transition function
- $\delta_{ext} : Q \times X^b \rightarrow S$ is the external transition function
  - $Q = \{(s,e) : s \in S, 0 \leq e \leq ta(s)\}$ is the total state set, where e is the time elapsed since last transition
  - X^b is a set of bags over elements in X
- δ_{con} is the confluent transition function, subject to $\delta_{con}(s, \oslash) = \delta_{int}(s)$
- $\lambda : S \rightarrow Y$ is the output function
- $ta(s) : S \rightarrow \Re_0^+ \cup \infty$ is the time advance function.

The formal definition of a coupled model is described as:

$$N = \langle X, Y, D, EIC, EOC, IC \rangle \tag{4}$$

where,

- X is the set of external input events
- Y is the set of output events
- D is a set of DEVS component models
- EIC is the external input coupling relation
- EOC is the external output coupling relation
- IC is the internal coupling relation.

The coupled model N can itself be a part of component in a larger coupled model system giving rise to a hierarchical DEVS model construction.

Fig. 1 shows a coupled DEVS model. M1 and M2 are DEVS models. M1 has two input ports: "in1" and "in2", and one output port: "out". The M2 has one input port: "in1", and two output ports: "out1" and "out2". They are connected by input and output ports internally (this is the set of internal couplings, IC). M1 is connected by external input "in" of Coupled Model to "in1" port, which is an external input coupling (EIC). Finally, M2 is connected to output port "out" of Coupled Model, which is an external output coupling (EOC).

The DEVSJAVA [1] is a Java based DEVS simulation environment. It provides the advantages of Object Oriented framework such as encapsulation, inheritance,

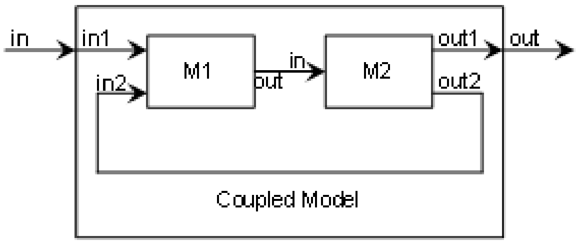


Fig. 1. Coupled DEVS model

and polymorphism. DEVSJAVA manages the simulation time, coordinates event schedules, and provides a library for simulation, a graphical user interface to view the results, and other utilities. Detailed descriptions about DEVS Simulator, Experimental Frame and of both atomic and coupled models can be found in [35].

2.3 DEVS/SOA

The Service oriented Architecture (SOA) is a framework consisting of various W3C standards, in which various computational components are made available as "services" interacting in an automated manner achieve machine-to-machine interoperable interaction over the network. Web-based simulation requires the convergence of simulation methodology and WWW technology (mainly Web Service technology). The fundamental concept of web services is to integrate software application as services. Web services allow the applications to communicate with other applications using open standards. We are offering DEVS-based simulators as a web service, which are based on these standard technologies: communication protocol (Simple Object Access Protocol, SOAP), service description (Web Service Description Language, WSDL), and service discovery (Universal Description Discovery and Integration, UDDI).

Fig. 2 shows the framework of our distributed simulation using SOA. The complete setup requires one or more servers that are capable of running DEVS Simulation Service. The capability to run the simulation service is provided by the server side design of DEVS Simulation protocol supported by the latest DEVSJAVA Version 3.1.

The Simulation Service framework is two layered framework. The top-layer is the user coordination layer that oversees the lower layer. The lower layer is the true simulation service layer that executes the DEVS simulation protocol as a Service. The lower layer is transparent to the modeler and only the top-level is provided to the user.

The top-level has three main services: upload DEVS model, compile DEVS model, and simulate DEVS model. The second lower layer provides the DEVS

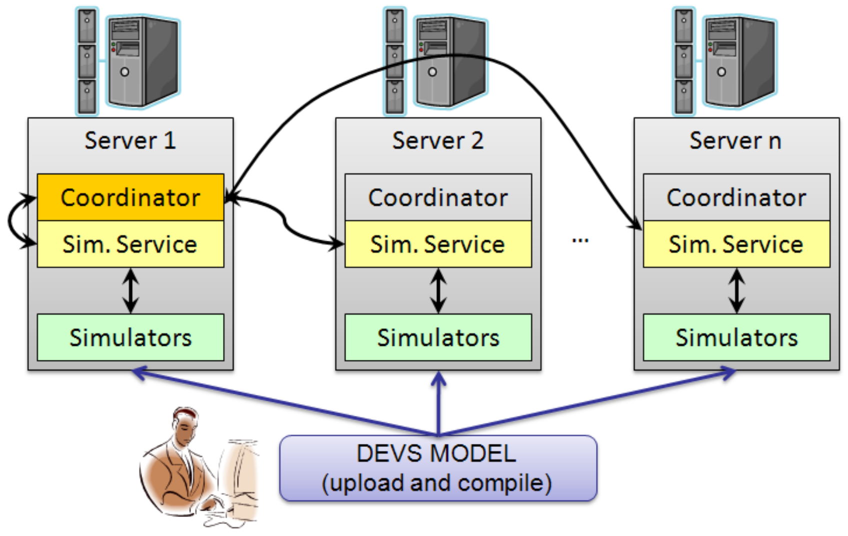


Fig. 2. DEVS/SOA distributed architecture

Simulation protocol services: initialize simulator *i*, run transition in simulator *i*, run lambda function in simulator *i*, inject message to simulator *i*, get time of next event from simulator *i*, get time advance from simulator *i*, get console log from all the simulators, and finalize simulation service.

The explicit transition functions, namely, the internal transition function, the external transition function, and the confluent transition function, are abstracted to a single transition function that is made available as a Service. The transition function that needs to be executed depends on the simulator implementation and is decided at the runtime. For example, if the simulator implements the *Parallel DEVS (P-DEVS)* formalism, it will choose among internal transition, external transition or confluent transition.

The client is provided a list of servers hosting DEVS Service. He selects some servers to distribute the simulation of his model. Then, the model is uploaded and compiled in all the servers. The main server selected creates a coordinator that creates simulators in the server where the coordinator resides and/or over the other servers selected. This whole framework is known as DEVS/SOA framework and details are available at [22], [21].

Summarizing from a user's perspective, the simulation process is done through three steps (Fig. 3): (1) write a DEVS model (currently DEVSJAVA is only supported), (2) provide a list of DEVS servers (through UDDI, for example). Since we are testing the application, these services have not been published using UDDI by now. Select *N* number of servers from the list available, and (3), run the simulation (upload, compile and simulate) and wait for the results.

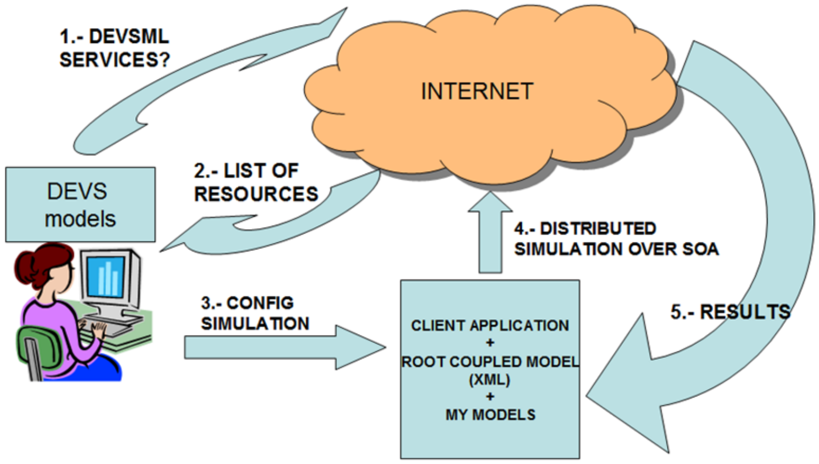

1.- DEVSML SERVICES?
INTERNET
DEVS models
2.- LIST OF RESOURCES
4.- DISTRIBUTED SIMULATION OVER SOA
3.- CONFIG SIMULATION
CLIENT APPLICATION + ROOT COUPLED MODEL (XML) + MY MODELS
5.- RESULTS

Fig. 3. Execution of DEVS SOA-Based M&S

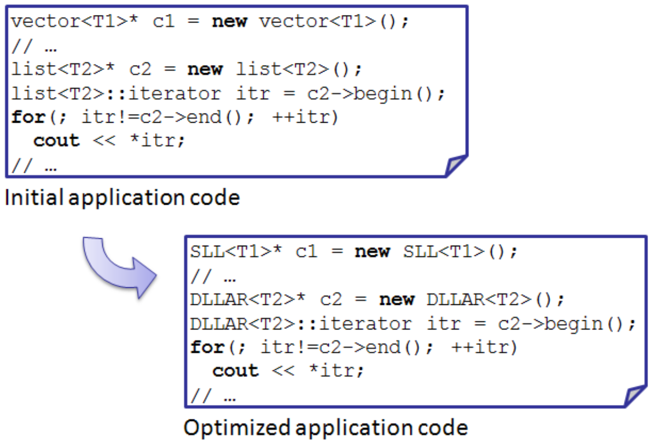
3 The Dynamic Data Types Exploration Problem

DDTs are software abstractions by means of which we can manipulate and access data. The implementation of DDT has two main effects on the performance of an application. First, it involves storage aspects that determine how data memory is allocated and freed at run-time, and how this memory is tracked. Second, it includes an access component, which can refer to two different basic access patterns: sequential (or iterator-based) and random access.

Fig. 4 shows an example of DDTs exploration. The initial code contains two containers, *c1* and *c2*, instantiated as a *vector* and a *list*, respectively. After the exploration process, we can obtain for example a candidate solution that recommends *c1* to be instantiated as *Single Linked List (SLL)* and *c2* as *Double Linked List of Arrays (DLLAR)*.

More generally we can state that the application to optimize contains a set of containers *C*, which are candidates to be instantiated as a certain DDT from the set of possible implementation of DDTs library *D* presented in [3] [10]. Thus, the goal of our optimization flow is to obtain a set of pairs (container, DDT) $\{c_i \in C, d_j \in D\}$, such that minimizes memory accesses, memory usage and power consumption for the target embedded system. Additional constraints as the minimum and maximum values for all three objectives may be defined. Clearly, this is a multi-objective optimization problem.

To measure the quality of a solution, we have defined the equations to evaluate the behavior of DDT implementations by means of parameters such as the number



```
vector<T1>* c1 = new vector<T1>();
// …
list<T2>* c2 = new list<T2>();
list<T2>::iterator itr = c2->begin();
for(; itr!=c2->end(); ++itr)
  cout << *itr;
// …
```

```
SLL<T1>* c1 = new SLL<T1>();
// …
DLLAR<T2>* c2 = new DLLAR<T2>();
DLLAR<T2>::iterator itr = c2->begin();
for(; itr!=c2->end(); ++itr)
  cout << *itr;
// …
```

Fig. 4. Code before and after the exploration of Dynamic Data Types

of sequential accesses, random accesses, average size, etc. In our case we have classified the DDT implementations in basic DDT and multi-layer implementations relevant for embedded multimedia applications. Table 2 contains the DDTs implemented [3].

Table 2. DDT library

DDT	Description
AR	Array
AR(P)	Array of pointers
SLL	Single-linked list
DLL	Doubly-linked list
SLL(O)	Single-linked list with roving pointer
DLL(O)	Doubly-linked list with roving pointer
SLL(AR)	Single-linked list of arrays
DLL(AR)	Doubly-linked list of arrays
SLL(ARO)	Single-linked list of arrays and roving pointer
DLL(ARO)	Doubly-linked list of arrays and roving pointer

Once we have fixed the problem optimization process for DDTs, we can describe the whole process shown in Fig. 5. It has three main steps: Profiling of the application, estimation of the parameters and multi-objective optimization algorithms execution. These three steps are described in the next sections.

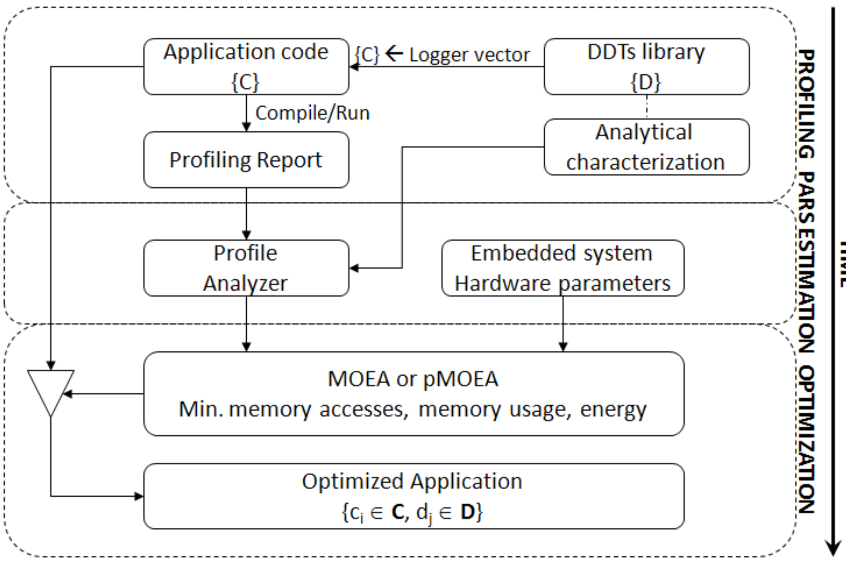


Fig. 5. DDTs optimization flow

3.1 Profiling of the Application

In order to evaluate the different metrics we need to obtain the real execution information from the application. Unfortunately, the execution of the whole application is not a viable solution. An alternative good solution recently proposed [10] is to obtain a profiling report of the application where the following information is logged: number and location of the accesses of an element, addition of an element, removal of an element, the clearing of the container, iterator operations such as pre-increment or dereference, constructor, destructor, copy constructor and swap operation. To this end, we need to replace all the candidate variables in the application by our vector DDT implementation, which logs all the information needed to evaluate them the using equations developed in [3].

3.2 Parameters Estimation

In this phase, we extract all information needed from the profiling report. The purpose is to measure the quality of a solution (c_i, d_j) in the DDT exploration, using several parameters, namely, the number of candidate variables, number of elements stored in the container in the worst case (N_e), average of the number of elements stored (N_{ve}), size of the elements in bytes (T_e), size of the pointers in bytes (T_{ref}), number of read accesses (N_r), number of write accesses (N_w) and cache misses (N_{pa}). All these parameters can be extracted from the profiling report. To this end, we have developed a tool called Profile Analyzer. Cache misses are also obtained by means of simulation, generating memory traces from the profiling report and the DDT library, using them as input for the Dinero IV cache simulator [13] for the particular memory configuration of the target embedded system. This phase is the most-time consuming part of the exploration, although it is done only once for each target architecture, and for each tested application.

3.3 Optimization

The last phase is the optimization process. It takes as input the parameters obtained in the previous phase and minimizes three objectives: memory accesses (MA), memory usage (MU) and energy (E), defined by the following equations, where Hw represents the effect that hardware parameters (memory architecture, CPU power, line sizes, memory access time, etc.) have on the optimization [25].

$$\begin{aligned} MA(\mathbf{c},\mathbf{d}) &= f_{MA}(N_e, N_{ve}, N_r, N_w) \\ MU(\mathbf{c},\mathbf{d}) &= f_{MU}(T_e, T_{ref}, N_e) \\ E(\mathbf{c},\mathbf{d}) &= f_E(N_r, N_w, N_{pa}, Hw) \end{aligned} \tag{5}$$

Memory accesses of the system f_{MA} is given by the following equation:

$$f_{MA} \propto N_e \times (N_r + N_w) + N_{ve} \tag{6}$$

The exact form of equation 6 depends on each DDT selected in $(\mathbf{c},\mathbf{d})$. It takes into account the number of random and sequential accesses to the elements stored in the DDT, as well as the number of creations and destructions of the container.

Memory usage f_{MU} is given by the following equation:

$$f_{MU} \propto T_{ref} + N_e \times \left(T_{ref} + T_e\right) \tag{7}$$

As in equation 6, the exact form of equation 7 depends on each DDT selected. It calculates the amount of memory used by each element stored in the DDT.

Finally, energy equation of the system is given by the following equation:

$$\begin{aligned} f_E = {} & t_{ex} \times CPU_{pow} + \\ & (N_r + N_w) \times (1 - N_{pa}) \times C_{accE} + \\ & (N_r + N_w) \times N_{pa} \times C_{accE} \times C_{lineS} + \\ & (N_r + N_w) \times N_{pa} \times DRAM_{accP} \times \\ & \left(DRAM_{accT} + \frac{C_{lineS}}{DRAM_{bandW}}\right) \end{aligned} \tag{8}$$

where t_{ex} is the system's total execution time, CPU_{pow} is the total processor power excluding the cache power, C_{accE} is the cache access energy, C_{lineS} is the cache line size, $DRAM_{accP}$ is the active power consumed by the DRAM, $DRAM_{accT}$ is the DRAM latency time, and $DRAM_{bandW}$ is the bandwidth of the DRAM.

There exist four components in the energy equation 8. The first term $t_{ex} \times CPU_{pow}$ calculates the processor energy given that execution time takes t_{ex} amount of time. The second term, $(N_r + N_w) \times (1 - N_{pa}) \times C_{accE}$ calculates the amount of energy consumed by the cache. The third term, $(N_r + N_w) \times N_{pa} \times C_{accE} \times C_{lineS}$ calculates the energy cost of writing to cache for each cache miss. The last term, calculates the energy cost of the DRAM to service all the cache misses.

The equation for calculating the system's total execution time t_{ex} is given by:

$$\begin{aligned} t_{ex} = {} & (N_r + N_w) \times (1 - N_{pa}) \times C_{accT} + \\ & (N_r + N_w) \times N_{pa} \times DRAM_{accT} + \\ & (N_r + N_w) \times N_{pa} \times \frac{C_{lineS}}{DRAM_{bandW}} + \\ & T_{bus} \end{aligned} \tag{9}$$

where C_{accT} is the access time of the cache.

There exist four components in the system's execution time shown in equation 9. The first term $(N_r + N_w) \times (1 - N_{pa}) \times C_{accT}$ is for calculating the amount of time taken for the processor to access the cache. The second term $(N_r + N_w) \times N_{pa} \times DRAM_{accT}$ calculates the amount of time required for the DRAM to respond to each cache miss. The third term calculates the amount of time taken to fill a cache line on each cache miss. The bus communication time cost is supposed to be constant (T_{bus}). As the bus communication time is expected to be similar to other systems, such decision will not adversely affect the final results.

Units for time variables in the equations are in seconds, bandwidth is in Bytes/sec., cache line size is in Bytes, power variable is in Watts, and energy unit is in Joules.

These equations are used by the optimization algorithm to evaluate the fitness of the solutions found in the exploration process. When the optimization process ends, it gives the DDT instantiation policy, i.e., which container should be implemented by which DDT. We also obtain the gain on memory accesses, memory usage and energy consumption.

3.4 Encoding a Solution

In order to apply a MOEA correctly we need to define a genetic representation of the design space of all possible DDT implementations alternatives. Moreover, to be able to cover all possible inter-dependencies of DDT implementations for different dynamic variables of an application, we must guarantee that all the individuals represent real and feasible solutions to the problem and ensure that the search space is covered in a continuous and optimal way [11].

Table 3 shows the representation of a chromosome. Genes are represented in the first row. Each of the chromosomes represents the set of DDT that should be used to instantiate all the corresponding containers in the application from Table 2.

For example, the second container $c_2 \in C$ will be instantiated by an array (AR). A chromosome contains n genes, where n is the number of the containers logged in the application, $n = size(C)$. We may use an integer to represent the values of a gene, and the constraint a gene must satisfy is: $1 \leq ddt \leq size(D)$.

Table 3. Example of an individual

Dynamic Data Type	*AR*	*AR*	*SLL*	...	*DLL*
Container	c_1	c_2	c_3	...	c_n

Consequently, if an application contains n containers, each individual (chromosome) has to be constituted by n integer fields (i.e., n genes). Our current implementation of the exploration framework optimizes up to 3128 variables using variations of the 10 possible DDTs contained in Table 2 for each of them. Thus, it can cover large real-life dynamic embedded applications.

4 Parallel Implementation

In this section we describe the parallel MOEA designed and how it is implemented in a DEVS environment.

4.1 pMOEA

We are employing pMOEAs for better performance when solving the exploration of DDTs in embedded applications described in Section 3, i.e., we are improving the quality of the solutions found and the time to obtain them. When developing pMOEA, some parameters must be defined [31]: MOEA(s) parallelized, topology, population size, migration rate, and replacement.

Regarding MOEAs and topology, we propose a coarse-grained pMOEA where each island may execute a different MOEA, in our case either NSGA-II [12] or SPEA2 [36]. We have used these two MOEAs because of their different complexity, but other algorithms could be included. SPEA2 is $O(N^3)$ and NSGA-II is $O(mN^2)$, where N is the population size and m is the number of objectives. Our islands are suited for a ring topology [5]. Experiments with other topologies are left for future study.

With respect to the population size of each island, few studies have been made in the literature[31]. For example, in [27], the sequential population is divided by the number of islands, remaining the size of the external set constant and equal to the initial population size. In this way, the number of islands increases, the execution time is reduced and the number of non-dominated solutions grows up. However, some metrics such as hypervolume or spread loose quality. In this work, we apply the following equation to the population size [26]:

$$P_i = \frac{P}{I} + \alpha \times \left(P - \frac{P}{I}\right) \tag{10}$$

where P_i is the population size of island $i \in [1..I]$, P is the population size in the sequential approach, $I > 1$ is the number of islands and $\alpha \in [0..1]$ is a scaling factor. Note that when I tends to infinity, P_i is constant: $\alpha \times P$. It is a design parameter that depends on the migration rate and hardware configuration (i.e. network bandwidth, processor types, etc). The purpose is to obtain better solutions in less computing time when the number of islands is increased. After several tests, we have set $\alpha = \frac{2}{I}$.

Regarding the external file size, we apply the following equation [26]:

$$P_i^E = P_i + N_I \tag{11}$$

where N_I is the total number of immigrants that island i will receive.

As in most of the pMOEAs, migration from one subpopulation to another is controlled by several parameters specified at the beginning of the execution and remains unchanged. These parameters are: (a) the topology defined by the connections between islands, a ring in our case; (b) a migration rate that controls how many individuals migrate; and (c) a migration interval that determines the migration frequency. Our migration rate is set to $P/100$, where P is the population size in the sequential algorithm. The best $P/100$ individuals are selected in the following way. First, we extract the set of non-dominated solutions in the current population P_i. Second, we sort the resulting set with respect to one random objective, and extract the first $P/100$ individuals. Moreover, since NSGA-II is faster than SPEA2 ($O(mN^2)$ vs. $O(N^3)$), NSGA-II could finish first while SPEA2 is still exploring early generations. Thus, our migration policy is synchronized every 100 generations.

4.2 DEVS and DEVS/SOA Implementation

Fig. 6 provides a scheme of the parallel procedure with two atomic models (top of the figure) and their execution over time (bottom of the figure). Each atomic model represents an island and includes two pair of *request, response* output and input ports. *Request* connections are used to ask for the best individual of the adjacent atomic model, and *response* connections are used to send this individual when available (every 100 generations, in Fig. 6). In other words, the specific MOEA (NSGA-II or SPEA2) is applied to each atomic model separately, and the best partial results are periodically sent from one atomic model to its neighbor on a ring communication topology.

We have implemented three variations that are tested in a multi-core and distributed architecture. The only difference between these variations is the MOEA algorithm that is controlling the subpopulation, i.e. running on each atomic model:

1. NS_K configuration: K atomic models executing NSGA-II and the same quantity running SPEA2, $2K$ islands in total.
2. SS_K configuration: $2K$ atomic models, but running all of them SPEA2 algorithm.
3. NN_K configuration: $2K$ atomic models using the NSGA-II algorithm.

The fitness function, the operators, and the stop criterion are the same as in the sequential version.

The algorithm shown in Fig. 6 follows a multi-threaded design, which is suitable to be executed in multi-core architectures. Another approach we have implemented consists of executing our proposed pMOEA in a set of workstations connected over a LAN. To this end, using our DEVS/SOA framework, we have executed 32 atomic models on 16 workstations each of two cores. The algorithm is exactly the same, but each workstation executes two atomic models. Individuals are sent between different workstations using web services [22]. Fig. 7 depicts an illustrative example of two workstations each running two MOEAs. Every workstation executes two MOEAs as a DEVS coupled model. The coupled models are connected in the desired topology (a ring in our case), which again is another design parameter that could impact the performance. Our atomic models are suited for a ring topology as well.

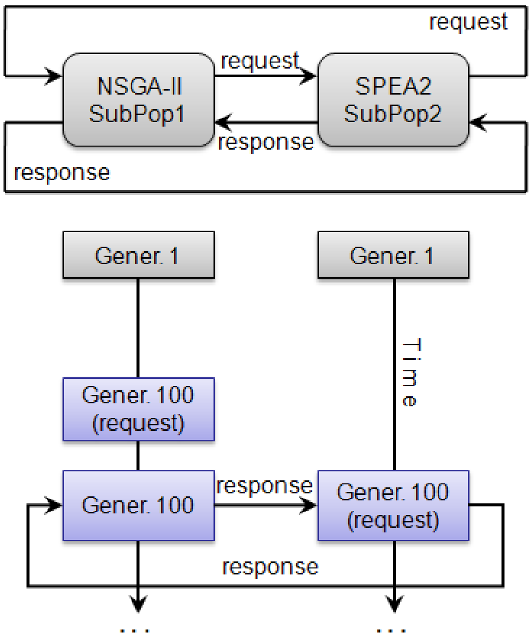


Fig. 6. A graphic representation of the DEVS model (multi-core architecture) and its evolution over time

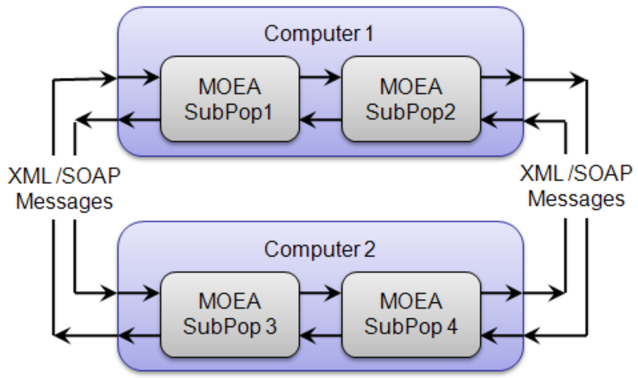


Fig. 7. A graphic representation of the DEVS model (multi-core/distributed architecture)

5 Experimental Methodology

In this section we describe the complete method applied to compare the different type of sequential and parallel MOEAs while optimizing a real-life dynamic embedded application.

We have evaluated the proposed optimization framework for a 3D Physics Engine for elastic and deformable bodies [19], which is a 3D engine that displays the interaction of non-rigid bodies. It includes 3128 dynamic containers in its source code for which we select the optimal DDT implementation in Table 2. It can cover all of the real-life embedded applications we are aware off.

5.1 *Embedded System HW/SW Specification*

The model of the embedded system architecture consisted of a processor with an instruction cache, a data cache, and embedded DRAM as main memory. The data cache uses a write-through strategy. The system architecture is illustrated in Fig. 8.

To analyze the effect of MOEAs on embedded system's memory accesses, memory usage and energy consumption, we utilized processor energy from [6], and the access time and energy values for caches of 32KB and embedded 16MB DRAM main memory from [29] and [17], respectively. The processor and memory specification is described in Table 4.

5.2 *Performance Metrics*

To compare the performance of different MOEAs, we need to evaluate the obtained set of non-dominated solutions considering: (1) Convergence to POF. (2) Diversity

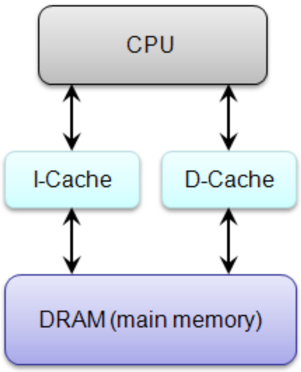


Fig. 8 System architecture: Instruction cache, data cache, and embedded DRAM as main memory

Table 4. System specification

Processor Energy	168mW, 100MHz
Embedded DRAM	100MHz
Energy	19.5 mW
Latency	19.5 ns
Bandwidth	50MB/s

on POF. Since the size of possible DDT implementations is large and it is not possible to cover the exact set of the POF, we compare the obtained *Pareto Front (PF)* with each other. In this direction, we select the following metrics to evaluate the performance of our approach.

5.2.1 Coverage

We use the coverage metric [37] to measure convergence. Let PF', PF'' be two sets of non-dominated solutions. The coverage metric can be defined as follows:

$$C(PF',PF'') = \frac{|p'' \in PF''; \exists p' \in PF' : p' \preceq p''|}{|PF''|} \tag{12}$$

The value $C(PF',PF'') = 1$ means that all points in PF'' are dominated by or equal to points in PF'. On the other hand, $C(PF',PF'') = 0$ means that no solutions in PF'' are covered by the set PF'. Both $C(PF',PF'')$ and $C(PF'',PF')$, have to be considered, since $C(PF',PF'')$ is not necessary equal to $C(PF'',PF')$. If $C(PF',PF'') > C(PF'',PF')$, the rate of dominated solutions in PF' is higher than in PF''.

5.2.2 Hypervolume or S-Metric

This metric calculates the volume (in the objective space) covered by members of a nondominated set of solutions Q [37]. Let v_i be the volume enclosed by solution $i \in Q$. Then, a union of all hypercubes is found and its hypervolume (H_V) is calculated.

$$H_V = \bigcup_{1}^{|Q|} v_i \tag{13}$$

The hypervolume of a set is measured relative to a reference point, usually the anti-optimal point or "worst possible" point in space. (We do not address here the problem of choosing a reference point, if the anti-optimal point is not known or does not exist one suggestion is to take, in each objective, the worst value from any of the fronts being compared). If a set X has a greater hypervolume than a set Y, then X is taken to be a better set of solutions than Y. Since this metric is not free from arbitrary scaling of objectives, we have evaluated the metric by using normalized objective function values.

5.2.3 Non-dominated Solutions

Given that DDTs optimization is a difficult problem, finding a high number of non-dominated solutions could be itself a hard challenge for any multi-objective optimizer. In this sense, the number of non-dominated solutions can be considered as a measure of the ability of the algorithm for exploring difficult search spaces.

We compare the obtained sets of non-dominated solutions by means of the above three criteria.

6 Experimental Results

To compare the performance of both sequential and parallel algorithms, the number of generations, and probability of crossover and mutation are set to the same values. After different tests, we have fixed them to the values indicated in Table 5. The sequential population size is set to 200 for each atomic model. In our parallel simulations, the population size follows equation 10. Migration rate and frequency are those described in Section 4. In all cases, the external archive size (where non-dominated solutions are stored) is set to the value given by equation 11.

Next, we summarize the results obtained by the sequential and parallel evolutionary algorithms. As it was mentioned in Section 4, we are able to run our MOEAs under three configurations: (1) a stand-alone atomic model (sequential architecture), (2) several atomic models running in separated threads (multi-core architecture) which utilize multiple processors when available, and (3) several atomic models running in separated threads and distributed amid a set of workstations (multi-core/distributed architecture). The distributed version is configured by using the DEVS/SOA framework. The experiments have been made using 16 workstations

Table 5. Parameters for evolutionary algorithms

Parameter	Value
Population size	200
Number of generations	8000
Probability of crossover	0.80
Probability of mutation	0.01

Intel® Core™ 2 CPU 6600 2.40GHz with 2GB DDR memory connected via 100Mbps Ethernet network.

6.1 Sequential DEVS Architecture

We have tested the sequential DDTs exploration speed in comparison to different alternative methods for the 3D Physics Engine application on a Intel® Core™ 2 CPU 6600 2.40GHz with 2GB DDR memory. Execution times are calculated by averaging results of 10 trials. The results obtained for the different tested exploration methods are shown in Table 6. We have compared our algorithms with state-of-the-art pruning and optimization methods for DDT implementations presented in [33], [10]. In these cases breadth-first, deep-first and branch & bound exploration heuristics are used to minimize overall memory access, memory usage and energy consumption in embedded multimedia applications. In this context, we have used a weighted sum of the three objectives as the fitness function for these three algorithms. Since there are 10^{3128} feasible solutions (10 DDTs for 3128 containers) it is unfeasible to reach the complete POF by means of exhaustive exploration. The results in Table 6 outline that the exploration process with our method (using NSGA-II and SPEA2) is much faster than using directly the implementations of DDTs and other heuristics, namely, 470× faster. Note that although in theory VEGA is faster than both NSGA-II and SPEA2, our design framework is able to obtain better speed-ups. This is because of our *Profile Analyzer* tool (Section 3), which can extract all

Table 6. Comparison between the proposed sequential algorithms and other techniques

Exploration method	Time (seconds)
Breadth-First	$11.23 \times 10^5 \pm 98.62$
Depth-First	$43.20 \times 10^4 \pm 87.13$
Branch&Bound	$10.80 \times 10^3 \pm 55.42$
VEGA [3]	$7.20 \times 10^3 \pm 103.20$
NSGA-II	$2.39 \times 10^3 \pm 0.78$
SPEA2	$3.83 \times 10^3 \pm 4.37$

the needed information from the profiling report, and it is done once for the target embedded application.

6.2 Multi-core DEVS Architecture

In order to exploit our 2-cores architecture, we have explored DDTs with some configurations of the three algorithms proposed (i.e., NN_K, NS_K and SS_K) on an Intel® Core™ 2 CPU 6600 2.40GHz with 2GB DDR memory. All the values presented are calculated by averaging results of 50 trials.

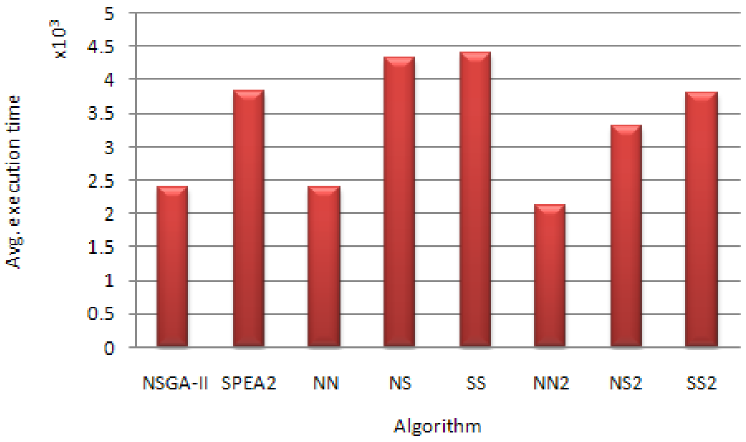
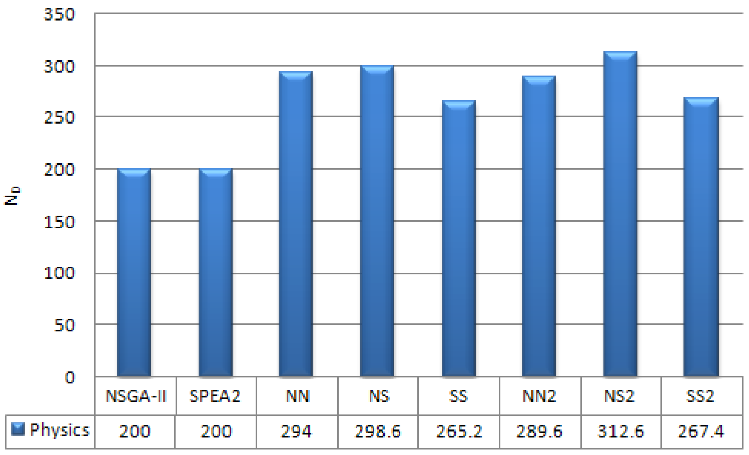


Fig. 9. Comparison between our sequential and multi-core algorithms

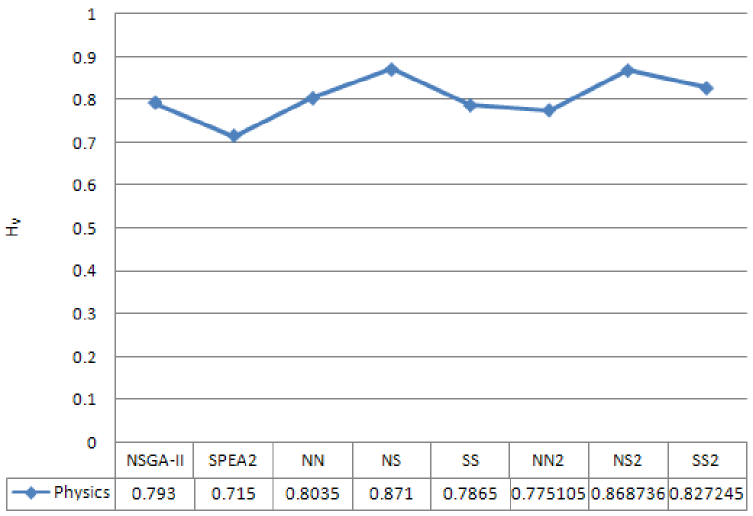
Fig. 9 shows the comparisons between the execution times of both sequential and parallel algorithms. Regarding pMOEAs, the number of islands is increased, the execution time is reduced. With respect to NN, SS and NS, we can see that the execution time is greater than in the sequential version. It is because equation 10 has been designed to balance the loss of non-dominated solutions when the number of islands grows up. However, as Fig. 9 depicts, NN_2 is faster than NSGA-II, and SS_2 is faster than SPEA2. To conclude, except in the case of two islands, all the pMOEAs are faster than the sequential version, even if more islands than cores are used (see SS_2 vs. SPEA2 in Fig. 9, for example). Between pMOEAs, the fastest one is NN_2, as each island uses the smallest population size with the fastest algorithm.

Fig. 10 depicts the number of non-dominated individuals obtained. NSGA-II offers the same non-dominated solutions as SPEA2. NS offers 49.3% more optimal solutions than both NSGA-II and SPEA2, and NS_2 4.69% more than NS. Thus, with respect to N_D, NS_K offers more optimal alternatives to the system designer for the implementation of the final embedded application.



	NSGA-II	SPEA2	NN	NS	SS	NN2	NS2	SS2
Physics	200	200	294	298.6	265.2	289.6	312.6	267.4

Fig. 10. Non-dominated individuals obtained by NSGA-II, SPEA2, NN_K, SS_K and NS_K, with $K = 1,2$



	NSGA-II	SPEA2	NN	NS	SS	NN2	NS2	SS2
Physics	0.793	0.715	0.8035	0.871	0.7865	0.775105	0.868736	0.827245

Fig. 11. Hypervolume or S-metric obtained by NSGA-II, SPEA2, NN_K, SS_K and NS_K, with $K = 1,2$

Fig. 11 shows the hypervolume or S-metric obtained. NS_K algorithms reach better values compared to the other MOEAs, sequential or parallel. Thus, the result set from NS_K algorithms is taken to be a better set of solutions than those obtained from other algorithms.

Table 7. Coverage metric

	NSGA-II	SPEA2	NN	NS	SS	NN_2	NS_2	SS_2	AVG
NSGA-II	–	0.0660	0.0959	0.0813	0.1132	0.1546	0.1208	0.1246	0.1081
SPEA2	0.2260	–	0.1719	0.1149	0.1684	0.1518	0.1100	0.1519	0.1564
NN	0.3030	0.2450	–	0.1476	0.1776	0.2122	0.1764	0.2130	0.2107
NS	0.3890	0.3180	0.4031	–	0.3213	0.2153	0.2207	0.2299	**0.2996**
SS	0.3440	0.3170	0.3223	0.1030	–	0.2404	0.1256	0.2677	0.2457
NN_2	0.3680	0.2810	0.2795	0.0693	0.2478	–	0.1092	0.1869	0.2202
NS_2	0.3620	0.3650	0.3131	0.1575	0.2429	0.2580	–	0.2897	**0.2840**
SS_2	0.3580	0.3570	0.2753	0.1723	0.2984	0.2937	0.1249	–	0.2685
AVG	0.3357	0.2784	0.2659	**0.1208**	0.2242	0.2180	**0.1411**	0.2091	–

Finally, Table 7 shows the coverage values obtained. Last row and last column show the averaged coverage over each column and each row, respectively. Regarding convergence comparisons, Table 7 shows that, in average, NS_K algorithms are better than any other algorithm. For example, $C_{avg}(NS,*) > C_{avg}(*,NS)$ is $0.2996 > 0.1208$ or $C_{avg}(NS_2,*) > C_{avg}(*,NS_2)$ is $0.2840 > 0.1411$. In the same way, $C_{avg}(NS,*) > C_{avg}(SS,*)$ is $0.2996 > 0.2457$ and $C_{avg}(*,NS) < C_{avg}(*,SS)$ is $0.1208 < 0.2242$.Thus, NS_K offers more optimal alternatives to the system designer for the implementation of the final embedded application.

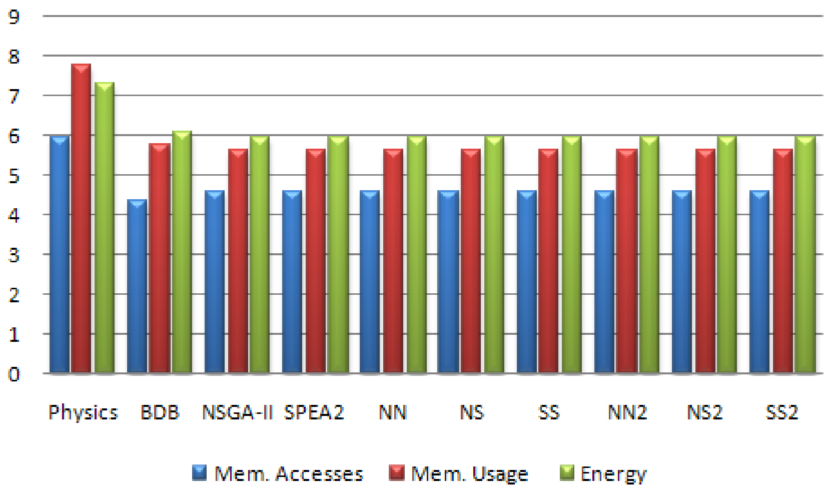


Fig. 12. Comparison of the real application with results obtained by our design framework (logarithmic scale).

For comparative reasons with the original application, we present Fig. 12 to illustrate the optimization process that our methodology performs. In this test, we compare the evaluation of our multi-objective function to the results obtained by all the algorithms used in our design framework. Since breadth-first, depth-first and branch & bound exploration methods offer the same solution, these results are grouped and labeled as BDB in Fig. 12. In the case of evolutionary algorithms, the set of solutions obtained is averaged. The figure shows the achieved level of optimization and final gains after applying the proposed design flow shown in Fig. 5. Furthermore, as this figure indicates, evolutionary algorithms offered the best compromise among objectives.

6.3 Multi-core DEVS/SOA Architecture

Finally, the NS_K configuration was distributed on a set of 16 workstations Intel® Core™ 2 CPU 6600 2.40GHz with 2GB DDR memory, connected via a 100Mbps Ethernet network. To this end, we placed two threads per workstation and the communication among workstations was made through our DEVS/SOA framework. All the values presented are calculated by averaging results of 10 trials.

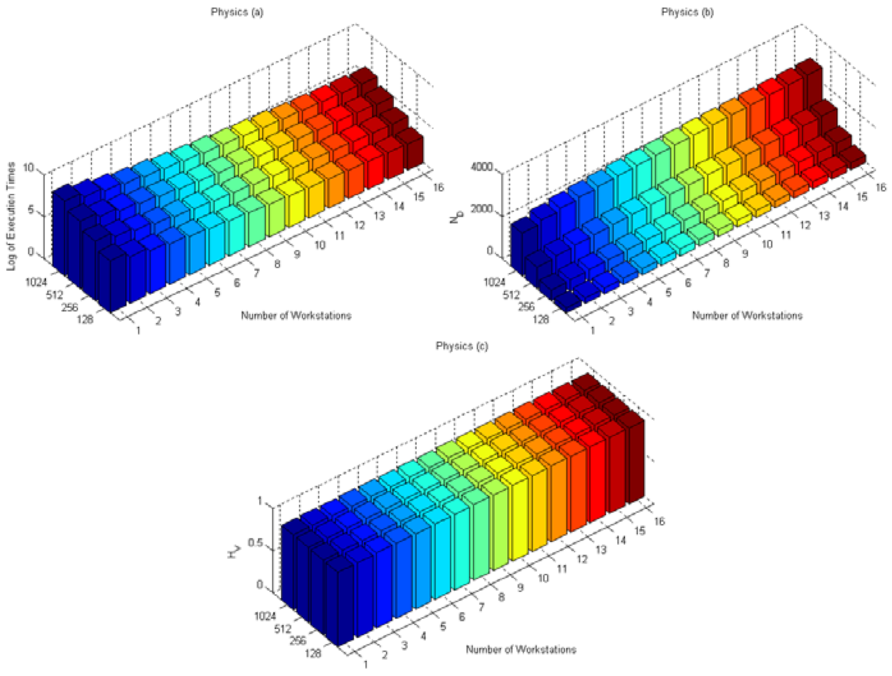


Fig. 13. Execution times (a), non-dominated solutions (b), and hypervolume (c) as a function of the number of workstations. Each workstation executes two DEVS atomic models

We tested our algorithm using from 1 to 16 workstations. This leads to 2, 4, 6, ..., 32 MOEAs running in parallel, namely NS_1, NS_2, NS_3, ..., NS_{16}, and different population sizes (128, 256, 512 and 1024). The tests were performed by changing only the number of workstations in order to observe and study the increase in performance (speed-up, N_D and H_V). In all these cases the number of generations was set to 8000. The population and external archive size of each island was set following equations 10 and 11, respectively.

In light of the results presented in Fig. 13, as the size of the population increased, the execution time of the parallel version improved proportionally to the number of islands (see Fig. 13a). Also, Fig. 13b indicates that the number of non-dominated individuals increased at logarithmic rate as the number of islands increased. Finally, as Fig. 13c depicts, the hypervolume remains constant along all the simulations, with non-significant variations.

This shows that the proposed pMOEA is better suited for large populations. It is also worthwhile to mention that with small populations, a parallel and distributed version of a genetic algorithm is most likely to converge to a local minimum due to a small gene pool.

7 Conclusions and Future Work

New multimedia embedded applications are increasingly dynamic, and rely on DDTs to store their data. The selection of optimal DDT implementations for each variable in a particular target embedded system is a very time-consuming process due to the large design space of possible DDTs implementations. In this research work we have studied several MOEAs to solve this problem. Particularly, we have proposed a new parallel algorithm (NS_K) which combines in a novel manner two widely used MOEAs. The problem is formulated as a multi-objective combinatorial optimization problem, for which we used three objective functions: memory accesses, memory usage and energy consumption. The results obtained shows that this parallel approach performs very well. In fact, NS_K reaches more optimal solutions than the other sequential and parallel algorithms, obtaining an execution time that decreases with the number of islands used.

We also have executed NS_K in a cluster of 16 workstations of two cores each. Our results show that if the size of the population is increased, the performance of the parallel version improves proportionally with respect to the number of available islands. As a result, we can conclude that not only parallel implementations improve the speed of the optimization process, but also the quality and the variety of the solutions, especially for large populations. Although we conducted our research experiments in a LAN setting, deploying the application over a grid enabled DEVS/SOA infrastructure allows us to capitalize on the speedup that we achieved in our proposed NS_K.

Future work includes the development of dynamic control parameters, such as, the topology, and a deeper study of migration rates and frequency. We are also working on exploring other alternatives with new combinations of different MOEAs to those used in this research work.

References

1. Arizona center of integrative modeling & simulation, acims (2008), http://www.acims.arizona.edu
2. Antonakos, J.L., Mansfield, K.C.: Practical Data Structures using C/C++. Prentice-Hall, Englewood Cliffs (1999)
3. Atienza, D., Baloukas, C., Papadopoulos, L., Poucet, C., Mamagkakis, S., Hidalgo, J.I., Catthoor, F., Soudris, D., Lanchares, J.: Optimization of dynamic data structures in multimedia embedded systems using evolutionary computation. In: SCOPES 2007: Proceedingsof the 10th international workshop on Software & compilers for embedded systems, pp. 31–40. ACM Press, New York (2007), http://doi.acm.org/10.1145/1269843.1269849
4. Benini, L., de Micheli, G.: System-level power optimization: techniques and tools. ACM Trans. Des. Autom. Electron. Syst. 5(2), 115–192 (2000), http://doi.acm.org/10.1145/335043.335044
5. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Dordrecht (2000)
6. Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P.G., Achteren, T.V., Omnes, T.: Data access and storage management for embedded programmable processors. Kluwer Academic Publishers, Dordrecht (2002)
7. Choi, Y., Kim, T., Han, H.: Memory layout techniques for variables utilizing efficient dram access modes in embedded system design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24(2), 278–287 (2005)
8. Coello, C.: A comparative survey of evolutionary-based multiobjective optimization techniques. Knowledge and Information Systems 1, 269–308 (1999)
9. Corne, D.W., Jerram, N.R., Knowles, J.D., Oates, M.J.: Pesa-ii: Region-based selection in evolutionary multiobjective optimization. In: Spector, L., Goodman, E.D., Wu, A., Langdon, W.B., Voigt, H.M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M.H., Burke, E. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), pp. 283–290. Morgan Kaufmann, San Francisco (2001)
10. Daylight, E.G., Atienza, D., Vandecappelle, A., Catthoor, F., Mendias, J.M.: Memory-access-aware data structure transformations for embedded software with dynamic data accesses. IEEE Transactions on VLSI Systems 12, 269–280 (2004)
11. Deb, K.: Multiobjective Optimization using Evolutionary Algorithms. John Wiley and Son Ltd., Chichester (2001)
12. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2), 182–197 (2002)
13. Edler, J.: Dinero iv trace-driven uniprocessor cache simulator (2008), http://pages.cs.wisc.edu/~markhill/DineroIV
14. Fernandez, J.M., Vila, P., Calle, E., Marzo, J.L.: Design of virtual topologies using the elitist team of multiobjective evolutionary algorithms. In: Obaidat, M., Gburzynski, P. (eds.) Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2007), San Diego, USA, pp. 266–271 (2007)
15. Fonseca, C.M., Fleming, P.J.: Genetic algorithms for multiobjective optimization: Formulation discussion and generalization. In: Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA 1993), pp. 416–423 (1993)
16. Hajela, P., Lin, C.Y.: Genetic search strategies in multicriterion optimal design. Structural Opt. 4, 99–107 (1992)

17. Hardee, K., Jones, F., Butler, D., Parris, M., Mound, M., Calendar, H., Jones, G., Aldrich, L., Gruenschlaeger, C., Miyabayashil, M., Taniguchi, K., Arakawa, I.: A 0.6v 205mhz 19.5ns trc 16mb embedded dram. In: IEEE International Solid-State Circuits Conference, ISSCC (2004)
18. Horn, J., Nafpliotis, N., Goldberg, D.E.: A niched pareto genetic algorithm for multi-objective optimization. In: Proceedings of the First IEEE Conference on Evolutionary Computation, vol. 1, pp. 82–87 (1994)
19. Kharevych, L., Khan, R.: 3d physics engine for elastic and deformable bodies. University of Maryland, College Park (2002), http://www.cs.umd.edu/Honors/reports/kharevych.html
20. Michalewicz, Z.: Genetic Algorithms + data structures = Evolution Programs. Springer, Heidelberg (1996)
21. Mittal, S., Risco-Martin, J.L., Zeigler, B.P.: Devs/soa: A cross-platform framework for net-centric modeling and simulation using devs. Submitted to SIMULATION: Transactions of SCS, in review (2007)
22. Mittal, S., Risco-Martín, J.L., Zeigler, B.P.: Devs-based web services for net-centric t&e. In: Summer Computer Simulation Conference, SCSC 2006 (2006)
23. Muttreja, A., Raghunathan, A., Ravi, S., Jha, N.K.: Automated energy/performance macromodeling of embedded software. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26(3), 542–552 (2007)
24. Panda, P.R., Catthoor, F., Dutt, N.D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., Kjeldsberg, P.G.: Data and memory optimization techniques for embedded systems. ACM Trans. Des. Autom. Electron. Syst. 6(2), 149–206 (2001), http://doi.acm.org/10.1145/375977.375978
25. Risco-Martin, J.L., Atienza, D., Hidalgo, J.I., Lanchares, J.: Analysis of multi-objective evolutionary algorithms to optimize dynamic data types in embedded systems. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2008 (2008)
26. Risco-Martin, J.L., Atienza, D., Hidalgo, J.I., Lanchares, J., Mittal, S.: Optimization of multimedia embedded applications using genetic algorithms and discrete event simulation over soa. Submitted to IEEE Transactions on Computer-Aided Design
27. Risco-Martín, J.L., Atienza, D., Hidalgo, J.I., Lanchares, J.: A parallel evolutionary algorithm to optimize dynamic data types in embedded systems. Soft Computing - A Fusion of Foundations, Methodologies and Applications 12(12), 1157–1167 (2008)
28. Schaffer, J.D.: Multiple objective optimization with vector evaluated genetic algorithms. In: Genetic Algorithms and their Applications: Proceedings of the First International Conference on Genetic Algorithms, pp. 93–100. Hillsdale, New Jersey (1985)
29. Shivakumar, P., Jouppi, N.P.: Cacti 3.0: An integrated cache timing, power, and area model. Tech. Rep. 2001/2, Compaq Computer Corporation (2001)
30. de Toro Negro, F., Ortega, J., Ros, E., Mota, S., Paechter, B., Martín, J.: Psfga: Parallel processing and evolutionary computation for multiobjective optimisation. Parallel Computing 30(5-6), 721–739 (2004)
31. Veldhuizen, D.A.V., Zydallis, J.B., Lamont, G.B.: Considerations in engineering parallel multiobjective evolutionary algorithms. IEEE Transactions on Evolutionary Computation 7(2), 144–173 (2003)
32. Wilson, L., Moore, M.: Cross-pollinating parallel genetic algorithms for multiobjective search and optimization. International Journal of Foundations of Computer Science 16(2), 261–280 (2005)
33. Wuytack, S., Catthoor, F., De Man, H.: Transforming set data types to power optimal data structures. IEEE Transactions on Computer-Aided Design 15(6), 619–629 (1996)

34. Xiong, S., Li, F.: Parallel strength pareto multi-objective evolutionary algorithm for optimization problems. In: Proceedings of the 2003 Congress on Evolutionary Computation (CEC 2003), vol. 4, pp. 2712–2718. IEEE Press, Canberra (2003)
35. Zeigler, B.P., Kim, T., Praehofer, H.: Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press, London (2000)
36. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In: Proceedings of the Evolutionary Methods for Design, Optimization and Control with Application to Industrial Problems, Barcelona, Spain, pp. 95–100 (2002)
37. Zitzler, E., Thiele, L.: Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. IEEE Transactions on Evolutionary Computing 3(4), 257–271 (1998)
38. Zydallis, J.B., Van Veldhuizen, D.A., Lamont, G.B.: A statistical comparison of multiobjective evolutionary algorithms including the MOMGA-II. In: Zitzler, E., Deb, K., Thiele, L., Coello Coello, C.A., Corne, D.W. (eds.) EMO 2001. LNCS, vol. 1993, pp. 226–240. Springer, Heidelberg (2001)

A Grid-Based Hybrid Hierarchical Genetic Algorithm for Protein Structure Prediction

Alexandru-Adrian Tantar, Nouredine Melab, and El-Ghazali Talbi

Abstract. A hybrid hierarchical conformational sampling evolutionary algorithm is presented in this chapter, relying on different parallelization models. After first reviewing general conformational sampling aspects, *e.g.* existing approaches, complexity matters, force field functions, a focus is considered for the protein structure prediction problem. Furthermore, having as basis the highly multimodal nature of the energy landscape structure, a hybrid evolutionary approach is defined, enclosing conjugate gradient and adaptive simulated annealing enforced components. An insular model is employed, the conformational sampling process being conducted on a collaborative basis. Nonetheless, although low energy conformations were obtained, no close to native conformations were attained. Consequently, a higher complexity hierarchical paradigm has been constructed, with incentive following results.

1 Introduction

Entitled as *a silent revolution* in a recollection of the last century preeminent discoveries [37, 21], contemporary computational biology extends over mathematical modeling, molecular biology and computer science, comprising inter-linked scientific research disciplines. *In silico* conformational modeling and simulation, although computationally expensive, ascertained significant advancements in the entire life sciences spectrum [50, 47]. Conclusive examples may be found by reminding the completion of the human genome mapping, attained this decade, Human Genome Project [56, 36], the Folding@Home project [42, 49] fighting cancer, and Alzheimer's disease, etc. Nonetheless, no advancement on the current state

Alexandru-Adrian Tantar · Nouredine Melab · El-Ghazali Talbi
INRIA Lille - Nord Europe, Project-Team DOLPHIN, Room 211 bis,
Building A, 40, Avenue Halley, Parc Scientifique de la Haute-Borne,
59655 Villeneuve d'Ascq Cedex, France
e-mail: `Alexandru-Adrian.Tantar@inria.fr`,
`{Nouredine.Melab,El-Ghazali.Talbi}@lifl.fr`

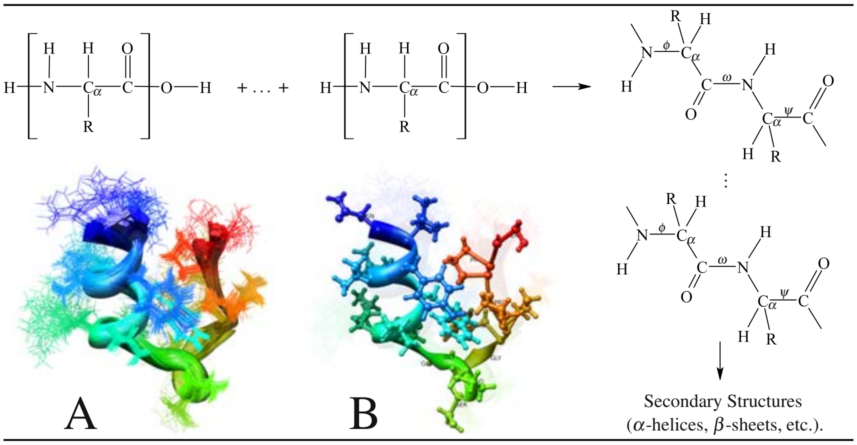


Fig. 1. First row: a NC$_\alpha$C back-bone structure resulting as a combination of multiple amino-acids; secondary and ternary structures follow. R designates the specific amino acid's *side chain* characteristic, ω, Φ and Ψ relate to dihedral angles. **Second row, A & B**: tryptophan-cage protein (PDB ID 1L2Y), multiple near-native conformations, respectively, a ribbon-ball&stick representation of a single conformation.

of the art is possible unless extensive grid computing is employed. At the core of avant-garde conformational sampling and molecular dynamics simulations, grid computing nowadays offers an unprecedented *sine qua non* computational support [20, 35], in this context, connecting the computational biology and computer science domains.

The foundations of this chapter address *ab initio* conformational sampling [40], having Protein Structure Prediction, further referred to as PSP, as a reference topic. Of particular interest for the parallel grid computing domain, the problem consists in determining the *ground-state* conformation of a specified protein, given its amino-acids sequence – the *primary structure*. In this context, the ground-state conformation term designates the associated tridimensional native form, referred to as *zero energy structure*. From a structural point of view, proteins are complex organic compounds composed of amino-acid residue chains joined by peptide bonds – for a graphical illustration, please refer to Fig. 1. Assenting to a concise definition, conformational sampling entails the exploration of an exponentially large space of possible configurations [41, 13, 9], derived on the basis of an extensive number of degrees of liberty, which define the flexibility of the under study conformation. An energetically stable configuration has to be computationally predicted with the support of an underlying, generally highly multimodal, force field function [45]. Of quintessential impact and reinforced by the *in vivo* realm ubiquitousness of proteins, the intrinsic relation connecting the structure of a protein and the corresponding biological function determines fundamental consequences for computer assisted drug design, the understanding of immune response mechanisms, etc. In addition, computational modeling and prediction offer an alternative to laboratory *in vitro* experimentation, unfeasible for large domain analysis.

For the herein presented conformational sampling study a paradigm combining an Evolutionary Algorithm (EA) and an Adaptive Simulated Annealing (ASA) technique is considered – to be further detailed in the following sections. A study comprising different local search algorithms and outlining the efficacy of the ASA method on several benchmark conformations was previously presented in [55]. In addition, an extensive analysis of different intensification and diversification operators has been presented in [54]. EAs are stochastic search iterative techniques, with a large area of application – epistatic, multimodal, multicriterion and highly constrained problems [8]. A direct subclass of the EAs, Genetic Algorithms (GAs) are Darwinian-evolution inspired, population-based metaheuristics that allow a powerful exploration of the conformational space. However, they have limited search intensification capabilities, which are essential for neighborhood-based improvement (the neighborhood of a solution refers in this context to a part of the problem's landscape). At the opposite extreme, the class of the different Simulated Annealing [34] algorithms presented in the literature, further denoted as *SA*s, offers weak ergodicity optimization techniques capable of dealing with multimodal functions of a large nonlinearity and discontinuity degree. Simulated annealing algorithms were developed by Kirkpatrick [34] as a generalization of the Metropolis Monte Carlo techniques [39], including as extension a temperature schedule which offers an improved control over the acceptance rate. The underlying paradigm simulates metal recrystallization in the process of annealing, the entropy of an initially disordered system being adiabatically reduced to low entropy states while maintaining at each step a thermodynamic equilibrium. The SAs represent a viable alternative to gradient based local search methods, being less prone to getting trapped in local minima. Furthermore, the implementation of an SA algorithm does not impose complex development constraints – as a counterpart and as opposed to EAs, simulated annealing techniques are extensively sequential in their nature thus being difficult to parallelize.

Furthermore, the currently available computational resources allow for higher complexity algorithmic constructions, rendering possible the design of hierarchical parallel and distributed approaches. Nonetheless, a complex algorithmic underlying layer has to be unfolded in order to effectively exploit the existing computational resources. A transparent deployment has to be ensured, endorsing large-scale distributed applications to be expanded over geographically dispersed clusters. The parallel construction of the here considered approaches is sustained by an MPI [23] based version of ParadisEO [7, 8], a framework dedicated to the reusable design of parallel hybrid meta-heuristics. A broad range of features is provided by the framework, including EAs support, local search methods, parallel and distributed models, hybridization mechanisms, etc. For a complete overview of the existing dedicated frameworks on parallel and grid specific metaheuristcs refer to [10, 8, 51, 1, 7].

The contents of this study inscribe in the context of *ANR Dock – Conformational Sampling and Docking on Computational Grids*[1], designated under the *Docking@Grid* acronym, a French National Research Agency three years funded project, scheduled to end by fall 2009. Encompassing distinct areas of expertise, the foundations of the project are set on the complementarity of the participant research teams and laboratories, specifically, (1) DOLPHIN, INRIA Lille – Nord Europe, Fundamental Computer Science Laboratory of Lille, LIFL, (2) Biology Institute of Lille, IBL CNRS/INSERM and (3) Life Sciences Division, CEA/iRTSV – Grenoble. As a final phase of the project, an *in vitro* biological validation of the attained results will be conducted under the competences of the Life Sciences Division, CEA. Note that all presented experimentations were performed on Grid'5000, a nation-wide computational grid, consisting of almost 5000 computational cores, shared in a network of nine academic centers. Conformational sampling results are reported on the basis of a large number of deployments, with up to almost 1000 computational cores.

The remainder of this chapter is organized as follows. An introduction discussing in brief protein structure prediction aspects is offered in Section 2, followed by an incremental presentation of the considered algorithmic components in Section 3. Encoding and evaluation function details are discussed, the formal basis of a conjugate gradient and of an adaptive simulated annealing algorithm being illustrated. A first hybrid parallel approach is afterwards introduced, implementation and execution environment details being also presented. As part of Section 4 the employed benchmark conformations are outlined, finally, experimental outcomes being discussed. As entailed by the drawn conclusions, a hierarchical parallel algorithm is proposed, addressing minima characterization issues – definition details and results are given. Conclusions and further directions are finally drawn.

2 Protein Structure Prediction

As outlined in the introduction, the PSP problem consists in determining the ground-state conformation of a specified protein, given its amino-acids sequence. The inter-atomic interactions to be considered for the protein structure prediction problem are a resultant of electrostatic forces, entropy, hydrophobic characteristics, hydrogen bonding, etc. Precise energy determination also relies on modeling solvent derived effects through dielectric constants and continuum model based terms – a more detailed, force field oriented discussion is presented in a following section. A trade-off is accepted in practice, opposing accuracy against the approximation level, varying from exact, physically correct mathematical formalisms to purely-empirical approaches. The main categories to be mentioned are *de novo, ab initio* electronic structure calculations, semi-empirical methods and molecular mechanics based models [16, 58, 40].

Accurate mathematical models, describing molecular systems, are formulated upon the *Schrödinger* equation [16], which makes use of molecular wavefunctions

[1] http://dockinggrid.gforge.inria.fr

for modeling the spatio-temporal probability distribution of the constituent entities. Nonetheless, although offering the most accurate approximation, the *Schrödinger* equation cannot be solved exactly for more than two interacting particles [16, 58]. At the opposite extreme, *empirical methods* rely upon molecular dynamics (*classical mechanics based methods*), and were introduced by Alder and Wainwright [2, 3]. Empirical methods do not make use of the quantum mechanics formalism, relying solely upon classical Newtonian mechanics, *i.e.* Newton's second law, and often represent the only applicable methods for large molecular systems, namely, proteins and polymers. After more than a decade protein simulations were initiated on bovine pancreatic trypsin inhibitor – BPTI [38].

Considering complexity aspects, as an example, for a reduced size molecule composed of 40 residues, a number of 10^{40} conformations must be taken into account when considering, in average, 10 conformations per residue. Furthermore, if a number of 10^{14} conformations per second is explored, a time of more than 10^{18} years is needed for determining the ground-state conformation. For example, for the *[met]-enkephalin* pentapeptide, composed of 75 atoms and having five amino-acids, *Tyr-Gly-Gly-Phe-Met*, and 22 variable backbone dihedral angles, a number of 10^{11} local optima is estimated. Detailed aspects concerning complexity matters were discussed in [13, 9]. As a conclusion, no simulation or resolution is possible unless extensive computational resources are used – it may be inferred that no polynomial time resolution is achievable if no or less *a priori* knowledge is employed.

For a comprehensive introductory article on the structure of proteins and related aspects please consult [40, 12]; a glossary of terms is also available in [57]. In addition, an extended referential resource for protein structural data may be accessed through the Brookhaven Protein Data Bank[2] [4].

3 A Parallel Hybrid Metaheuristic for the PSP

The exploration and intensification capabilities of the EAs do not suffice as paradigm, when addressing rough molecular energy function landscapes. Small variations of a torsional angle value may generate extremely different individuals, with respect to the fitness function. As a consequence, a nearly optimal configuration, considering the torsional angle values, may have a high energy value, and thus, it may not be taken into account for the future iterations of the algorithm. In order to correct the above exposed problem, a local search based method may be applied as a refinement step, alleviating the drawbacks determined by the conformation of the landscape – thus, a *Lamarckian* optimization technique is constructed.

3.1 Encoding of the Conformations and the Force Field Function

The algorithmic resolution of the PSP, in heuristic context, is directed through the exploration of the molecular energy surface. The sampling process is performed

[2] http://www.rcsb.org – Brookhaven Protein Data Bank; offers geometrical structural data for a large number of proteins.

by altering the structure of the under study conformation, *i.e.* backbone structure, associated torsional angles, etc., in order to obtain different structural variations. With implications over the sampling methodology, different encodings have been mentioned in literature. The trivial approach would consist of using a direct coding of the atomic Cartesian coordinates [46]. Nonetheless, as a main disadvantage of direct encoding based representations, filtering and correcting mechanisms are required, inducing a non-negligible overhead. Different other models were developed, including, for example, all-heavy-atom coordinates, C_α coordinates or backbone and residue atoms coordinates representations, hydrophobic/hydrophilic models [15], etc. For the herein described method, an indirect, less error-prone, torsional angle based representation has been preferred. More specifically, each conformation is coded as a vector of torsional angle values, denoted in the following as γ, $\gamma \overset{def}{=} (\gamma_1, \gamma_2, \cdots, \gamma_N)$, $\alpha_i \prec \gamma_i \prec \beta_i$, where N represents the liberty degree of the conformation and α_i, β_i stand as the lower and upper limits of the γ_i encoding value, $1 \le i \le N$. For a graphical illustration, please refer to Fig. 2.

$$
\begin{aligned}
E = & \sum_{bonds} K_b(b-b_0)^2 \\
+ & \sum_{angles} K_\theta(\theta-\theta_0)^2 \\
+ & \sum_{torsions} K_\phi(1-\cos n(\phi-\phi_0)) \\
+ & \sum_{Van\ der\ Waals} \frac{K^a_{ij}}{d^{12}_{ij}} - \frac{K^b_{ij}}{d^6_{ij}} \\
+ & \sum_{Coulomb} \frac{q_i q_j}{4\pi\varepsilon d_{ij}} \\
+ & \sum_{desolvation} \frac{K q_i^2 V_j + q_j^2 V_i}{d^4_{ij}}
\end{aligned}
$$

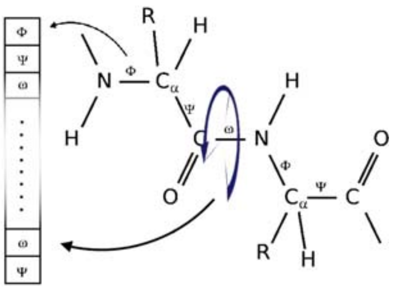


Fig. 2. Scoring function quantifying the inter-atomic interactions

The energy function, hereafter noted as E, is defined by relying on an independently calibrated *Consistent Valence Force Field (CVFF)* [14] based force field. The quantification of energy is performed by using empirical molecular mechanics, as depicted in Fig. 2. As classically employed for empirical force field definitions, a set of specific constants is associated with each interaction type, here denoted by K_b, K_θ, K_ϕ and K^a_{ij} for, respectively, bonds, angles, torsional angles and van der Waals interactions. An optimal value for the considered entity (bond, angle, torsion) is introduced through a corresponding $(A - A_0)$ equation term, where A, A_0 specify the sampled value, respectively the *a priori* experimentally determined optimal value. More specific, for the herein example, b represents bond lengths, θ angular values, ϕ torsional angles and q_a, d_{ij} and V_p the electrostatic charge associated to given atoms, the distance between the i and the j atoms, respectively, a volumetric

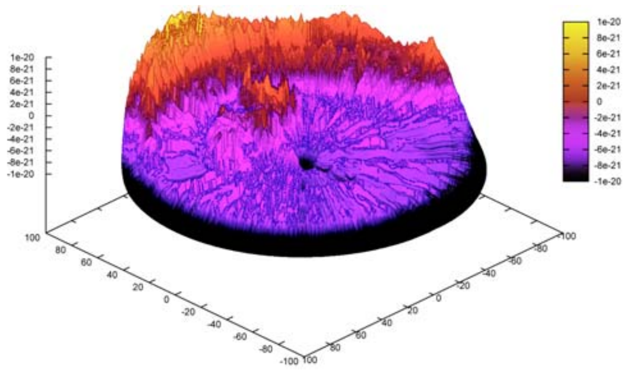


Fig. 3. Free energy surface for the *tryptophan-cage* protein around a deep optimum conformation. High energy points are depicted in light colors, the low energy points resulting in darker areas.

measure for the p atom. No further details are here included as being out of scope for the herein study – please refer to [45] for additional information.

An example of free energy surface representation for the *tryptophan-cage* is given in Fig. 3. The lighter areas of the surface correspond to high-energy conformations. The sampling values used for constructing the representation were computed by employing the Gibbs free energy over an ensemble of locally sampled conformations E_i – refer to Horvath *et al.* [26] for additional references and details:

$$G = -kT\left[\sum \exp\left(-\frac{E_i}{kT}\right)\right]$$

where $k = 1.3806504(24) \times 10^{23}\, J\, K^{-1}$ designates the constant of Boltzmann, offering, in numerical form, a connection between the molecular level and macroscopic observed effects, expressed as an ensemble result. Further, T represents a temperature term, the ensemble being equivalent to approx. 0.6 $kcal\ mol^{-1}$ at $300K$ – introductory notions and references were presented in [26].

An extensive discussion reviewing the force fields designed for protein simulations, with in-depth details, is offered in [45]. The first part of the study covers the evolution of the force fields over the last three decades, discussing various formulations which include the *Amber*, *CHARMM* and *OPLS* force fields.

3.2 Conjugate Gradient Local Search

The conjugate gradient method, an extension of the steepest gradient descent method, has been independently developed in the early 1950's by Eduard Stiefel and Magnus R. Hestenes, with the cooperation of J.B. Rosser, G. Forsythe and L. Paige [25]. Depending on the setup of the parameter values, the method converges to the closest local minimum, hence not being well adapted for the global optimization of large highly multimodal functions. References of early works on more advanced conjugate gradient methods lead to the publications of R. Fletcher, C.M. Reeves, M.J.D. Powell, E. Polak and G. Ribière [17, 18, 44], appeared a decade later.

For the rest of this section a *nonlinear* conjugate gradient approach is considered, simply referred to as *conjugate gradient*. If the force field based energy function E is continuous and differentiable in $\gamma_i \in \gamma, 1 \leq i \leq N$, the ∇E gradient is defined as a vector of partial derivatives:

$$\nabla E = \left[\frac{\partial E}{\partial \gamma_1} \; \frac{\partial E}{\partial \gamma_2} \; \cdots \; \frac{\partial E}{\partial \gamma_N} \right]^T \tag{1}$$

Hence, considering an iterative approach, at each iteration, the current point γ^+ can be updated by setting $\gamma^+ \leftarrow \gamma^+ - \tau_\varepsilon \nabla E_{\gamma^+}$, where the τ_ε step has a *positive, small enough value, adapted for the function under study*, and where ∇E_{γ^+} denotes the gradient vector computed at the γ^+ solution point. Compared to the steepest descent method, the conjugate gradient algorithm considers not only the gradient vector at the current point but also the previous directions. Hence, at each iteration k of the algorithm, the $\gamma^+_{\{k\}}$ solution is updated as follows:

$$\gamma^+_{\{k+1\}} \leftarrow \gamma^+_{\{k\}} - \tau_\varepsilon \delta_k, \; \delta_k \overset{def}{=} \begin{cases} \nabla E_{\gamma^+_{\{k\}}}, \text{ if } k = 0 \\ \nabla E_{\gamma^+_{\{k\}}} - \xi_k \delta_{k-1}, \text{ if } k > 0 \end{cases} \tag{2}$$

The algorithm is mainly based on the ξ_k factor which, in terms of convergence, defines the behavior of the method. Classically employed forms of the ξ_k term are defined as a combination of the previously computed gradient vectors, including different formulations, *e.g.* Fletcher-Reeves, Hestenes-Stiefiel, Polak-Ribière, etc. [17, 18, 44].

The basic pseudo-code of the nonlinear conjugate gradient method is given in Algorithm 1. The first step of the algorithm, for $k = 0$, is similar to the steepest descent method, the following steps relying in addition on the previously computed gradient vectors. For the herein example, the *Fletcher-Reeves* form has been chosen for the ξ_k term. Further, having computed the ξ_k, δ_k terms (lines 3-8), a line search is applied in order to minimize $E(\gamma^+_{\{k\}} - \tau_\varepsilon \delta_k)$ by varying the τ_ε factor. For details on line search algorithms refer to [48]. Different stopping criteria can be chosen – common approaches consider an *a priori* specified threshold for the gradient vectors (*e.g.* the absolute value of all the components of the $\nabla E_{\gamma^+_{\{k\}}}$ gradient vector falling below 1.0e-5) or for the attained improvement. In addition, a maximum number of iterations can be imposed.

The here employed component relies on *analytical gradient formulation*, the exploration being conducted on fine-grain landscape information. As a consequence, the method may not be well adapted for dealing with the conformational sampling landscape particularities, offering nevertheless fine-tuning minimization advantages.

Algorithm 1. Nonlinear Conjugate Gradient Pseudo-Code.

```
1: Set $k \leftarrow 0$, $\gamma^+_{\{k\}} \leftarrow \gamma$ ($\gamma$, $\gamma^+_{\{k\}}$ represent the current and the best known solution at iteration $k$, respectively)
2: repeat
3:     if $k = 0$ then
4:         Set $\delta_k \leftarrow \nabla E_{\gamma^+_{\{k\}}}$
5:     else
6:         Set $\xi_k \leftarrow \frac{\nabla E^T_{\gamma^+_{\{k\}}} \nabla E_{\gamma^+_{\{k\}}}}{\nabla E^T_{\gamma^+_{\{k-1\}}} \nabla E_{\gamma^+_{\{k-1\}}}}$
7:         Set $\delta_k \leftarrow \nabla E_{\gamma^+_{\{k\}}} - \xi_k \delta_{k-1}$
8:     end if
9:     Find $\tau_\varepsilon$ minimizing $E(\gamma^+_{\{k\}} - \tau_\varepsilon \delta_k)$
10:    Set $\gamma^+_{\{k+1\}} \leftarrow \gamma^+_{\{k\}} - \tau_\varepsilon \delta_k$
11:    Set $k \leftarrow k+1$
12: until $|E_{\gamma^+_{\{k\}}} - E_{\gamma^+_{\{k-1\}}}| < \tau_{prec}$ or $\nabla E_{\gamma^+_{\{k\}}} < \tau_{lb}$.
```

3.3 Adaptive Simulated Annealing Algorithm

Classical SA algorithms [34] rely on a Boltzmann sampling distribution, including as components a probability density function of the state space, $g(\gamma)$, an acceptance probability function $h(\Delta E)$ and an annealing schedule $T(k)$. Gradient information is not employed in classical constructions of the algorithm. The annealing schedule is defined over a number of discrete steps. The acceptance function has the role of quantifying the probability of performing a transition from an E_k energy state to a new state with energy E_{k+1}. Classical definitions make use of the Metropolis criterion [39] which makes use of the Boltzmann probability density function:

$$h(\Delta E) = \frac{e^{-E_{k+1}/T}}{e^{-E_{k+1}/T} + e^{-E_k/T}} = \frac{1}{1+e^{\Delta E/T}} \cong e^{-\Delta E/T},\ \Delta E = E_{k+1} - E_k \qquad (3)$$

Given a Gaussian-Markov system, with a probability density state space function $g(\Delta\delta) = (2\pi T)^{-N/2} e^{-\|\Delta\delta\|^2/(2T)}$, for an appropriate initial temperature T_0, the global minimum can be found if the temperature is decreased no faster than $T(k) = T_0/\ln k$. Low discrimination between solutions is considered in the initial phases of the algorithm, the method acting like a global search exploration. Near the final phases, local search is performed at low temperatures. Nonetheless, the main difficulty in designing a Boltzmann SA consists in determining the starting temperature as well as an efficient schedule for the problem under study. In practice, a $T_0/\ln k$ schedule does not offer a fast enough annealing. While no longer guaranteeing asymptotic convergence, exponentially decreasing schedules are preferred instead, *e.g.* $T(k) = e^{((c-1)k)} T_0, T(k) = c\ T(k-1), k \geq 1$, with $0 \ll c < 1$, $c \approx 0.98$.

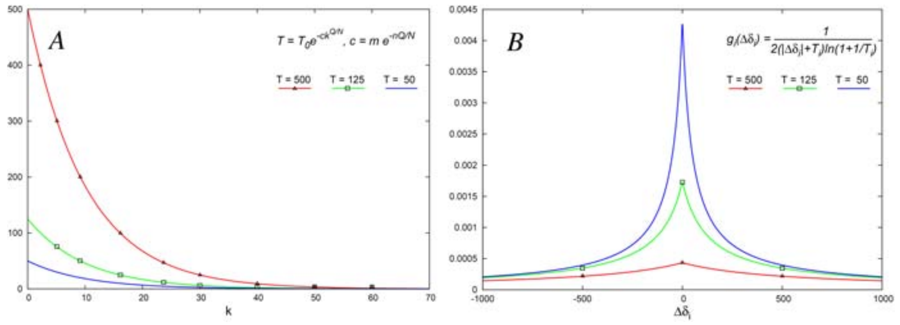


Fig. 4. A – Temperature decrease as defined for the ASA T_i schedules. **B** – ASA probability density function.

The Adaptive Simulated Annealing (ASA), an enhanced version of the basic SA algorithm, has been initially presented in the work of Ingber [27, 28, 31, 29, 30]. ASA exploits the characteristics of a specifically designed generating function allowing for an exponentially faster annealing process, as compared to the classical Boltzmann distribution based approach. In addition to employing a temperature parameter for the acceptance function, hereafter noted as T_a, distinct T_{ik_i} parameters and probability density functions are associated to each of the control parameters.

In the following, for simplicity, T_{ik_i} is denoted as T_i, with T_{i0} representing the initial temperature of the T_i schedule. As detailed in Ingber's articles, by considering $T_i \overset{def}{=} T_{i0}e^{(-c_i k_i^{Q_i/N})}$, with $c_i = m_i e^{-n_i Q_i/N}$, asymptotic convergence is attained. The m_i, n_i control parameters can be employed for adjusting and fine-tuning the algorithm for a specific problem. While for $Q_i > 1$ (quenching factors) an accelerated exploration is performed, the asymptotic convergence proof no longer stands, the algorithm being prone to getting trapped in local minima.

The adaptive features of the algorithm are determined by sensitivity derived factors, namely the T_a, T_i temperature schedules, which are employed in deciding over and controlling the acceptance, respectively, generation of new solutions – refer to Fig. 4 for a graphical depiction. The considered factors enclose descriptive information over the structure of the landscape to explore. The ASA generation function is defined over a set of uniform random variables, $u_i \in U[0,1]$, as exposed below:

$$\gamma_i^{k+1} = \gamma_i^k + \delta_i(\beta_i - \alpha_i), \text{ where } \delta_i \text{ is defined as:} \tag{4}$$

$$\delta_i = sgn(u_i - 0.5)\ T_i[\ (1 + 1/T_i)^{|2u_i - 1|} - 1\ ], \delta_i \in [-1, 1] \tag{5}$$

In the herein context α_i, β_i denote the lower, respectively upper limit of the γ_i encoding value. Acceptance is performed according to the Metropolis criterion. After a specified number of accepted solutions, *reannealing* takes place, adjusting the algorithm's parameters. Gradient based sensitivities are used for updating the acceptance temperature, T_a, the T_i temperature schedules and the k_i step indexes. No restriction is imposed on defining different sensitivity measures, other than gradient based ones. Considering γ^+, γ_a the best known solution and the last accepted solution, respectively, at a given step of the algorithm, for each component $\gamma_i^+ \in \gamma^+$, the associated sensitivity s_i is computed, to be employed in the reannealing step:

$$s_i = \left| \frac{\partial E}{\partial \gamma_i^+} \right|, \; s_{max} = \max_{1 \le i \le N} s_i, \; T_i' = \frac{s_{max}}{s_i} T_i, \; k_i' = \left[\frac{\ln(T_{i0}/T_i')}{c_i} \right]^N \tag{6}$$

$$T_{a0} = E(\gamma_a), \; T_a = E(\gamma^+), \; k = \left[\frac{\ln(T_{a0}/T_a)}{c} \right]^N \tag{7}$$

The main phases of the ASA method are depicted hereafter in Algorithm 2. The quenching factors Q, Q_i, the initial temperatures T_{a0}, T_{i0}, the m_i, n_i control parameters and the k, k_i step indexes are initialized in the first two lines. Lines 3 and 4 set the best known and the last accepted solutions which, for this step, are identical to the initial solution. The algorithm includes a main exploration loop (lines 5-32) and a secondary internal loop for generating new solutions (lines 6-11). Newly generated solutions are accepted based on the Metropolis criterion (line 13), the reannealing of the temperature schedules (lines 18-24) being performed at a pre-specified number of accepted solutions. At the end of each iteration of the main loop, step indexes and temperature schedules are updated in order to reflect the advancement of the algorithm (lines 26-31). The algorithm finishes after a fixed number of iterations or at a pre-specified threshold of iterations with no improvement.

As opposed to classical SA algorithms, the influence of the initial parameters over the exploration is alleviated, the annealing schedule being adaptively modified as to reflect the current exploration stage. While not directly employed in generating new solution points, gradient information is used for modifying the factors which intervene in the sampling process, consequently avoiding the direct disadvantages of steepest descent gradient based approaches. An improved scaling is offered as factors are independently modified on each dimension.

As a final remark, although including adaptive mechanisms, a large number of fine-tuning parameters are included. The effective calibration of the algorithm does not stand simplified tractableness basis, demanding for advanced parameter optimization. A possible approach, as suggested by Ingber, consists in using the ASA algorithm *per se* as a control parameters optimization component. Nevertheless, considering that performance evaluations require for the algorithm to be executed on one or multiple benchmarks, a high computational impact is implied. Subsequently, parallel support is required, entailing the optimization process to be carried on the support of a scalable distributed algorithm. Therefore, as part of the herein work, a meta-evolutionary algorithm has been employed in order to answer the mentioned concerns [53], given that ASA does not comport a high parallelization affinity.

A detailed description of the ASA algorithm, including comparison, test case studies and applications is available in the work of Ingber [30, 29, 27, 28, 31].

Algorithm 2. ASA Pseudo-Code.

```
1: Set $c$, $Q$, $k = 0$, $T_{a0} = E(\gamma)$
2: Set $Q_i$, $m_i$, $n_i$, $c_i = m_i e^{-n_i Q_i/N}$, $k_i = 0$, $T_{i0} = 1.0$, for $1 \le i \le n$
3: Set $\gamma^+ \leftarrow \gamma$ ($\gamma$, $\gamma^+$ represent the current and the best known solution, respectively)
4: Set $\gamma_a \leftarrow \gamma$ ($\gamma_a$ represents the last accepted solution)
5: **repeat**
6:     **for all** $\gamma_i \in \gamma,\ 1 \le i \le N$ **do**
7:         **repeat**
8:             $\delta_i \leftarrow sgn\left(u_i - \frac{1}{2}\right)\ T_i \left[ \left(1 + \frac{1}{T_i}\right)^{|2u_i - 1|} - 1 \right],\ u_i \in U[0,1]$
9:             $\gamma_i' \leftarrow \gamma_i + \delta_i(\beta_i - \alpha_i)$
10:        **until** $\alpha_i < \gamma_i' < \beta_i$
11:    **end for**
12:    $\Delta E \leftarrow E(\gamma') - E(\gamma)$
13:    **if** $u < e^{-\Delta E/T_a},\ u \in U[0,1]$ **then**
14:        Accept $\gamma'$ as the current solution: $\gamma \leftarrow \gamma'$, $\gamma_a \leftarrow \gamma'$
15:        **if** $E(\gamma') < E(\gamma^+)$ **then**
16:            Update the best known solution: $\gamma^+ \leftarrow \gamma'$
17:        **end if**
18:        **if** reannealing limit reached **then**
19:            **for all** $k_i,\ T_i,\ 1 \le i \le N$ **do**
20:                $s_i \leftarrow \left| \frac{\partial E}{\partial \gamma_i^+} \right|,\ \gamma_i^+ \in \gamma^+,\ s_{max} \overset{def}{=} \max_{1 \le i \le N} s_i$
21:                $T_i \leftarrow \frac{s_{max}}{s_i} T_i,\ k_i \leftarrow \left[ \frac{\ln(T_{i0}/T_i')}{c_i} \right]^N$
22:            **end for**
23:            $T_{a0} \leftarrow E(\gamma_a)$, $T_a \leftarrow E(\gamma^+)$, $k \leftarrow \left[ \frac{\ln(T_{a0}/T_a)}{c} \right]^N$
24:        **end if**
25:    **end if**
26:    **for all** $k_i,\ T_i,\ 1 \le i \le N$ **do**
27:        $k_i \leftarrow k_i + 1$
28:        $T_i \leftarrow T_{i0} e^{(-c_i k_i{}^{Q_i/N})}$
29:    **end for**
30:    $k \leftarrow k + 1$
31:    $T_a \leftarrow T_{a0} e^{(-ck^{Q/N})}$
32: **until** stopping criterion met.
```

3.4 Hybrid Parallel Genetic Algorithm

Evolutionary algorithms rely on a set of intensification vs. diversification directed operators for iteratively evolving an initial randomly generated population. At each iteration of the algorithm (generation), a selection process is conducted, the fitness of each individual being evaluated on a problem specific fitness function, *i.e.* the force field function for the herein case. The pseudo-code in Alg. 3 exposes the generic structure of an EA. Following a broad classification perspective, the main

Algorithm 3. EA Pseudo-Code.

```
$t \leftarrow 0$
Generate($P(0)$)
**while** $\neg$Termination_Criterion($P(t)$) **do**
    Evaluate($P(t)$)
    $P'(t) \leftarrow$ Selection($P(t)$)
    $P'(t) \leftarrow$ Apply_Reproduction_Ops($P'(t)$)
    $P(t+1) \leftarrow$ Replace($P(t), P'(t)$)
    $t \leftarrow t+1$
**end while**
```

subclasses of EAs are the Genetic Algorithms (GAs), Evolutionary Programming, Evolution Strategies, etc. In this context, a *genotype* represents the raw encoding of the individuals while the *phenotype* offers the equivalent representation features. At each generation, the genotype of a selected set of individuals is altered by applying mutation and crossover operators in order to intensify the exploration over an interest region or for diversification purposes as to avoid a premature convergence. Last, offsprings are reinserted in the population according to a pre-specified criterion.

The herein considered GA was parallelized in a hierarchical manner, including, in addition to the exposed basic pseudo-code, three levels of parallelism – the insular model, the parallel evaluation of the population and the synchronous multi-start model. A conceptual simplistic depiction of the different models is offered in Fig. 5. At execution time, a set of identically configured algorithms is deployed, independently evolving a local assigned population whereas fitness evaluations are dispatched on remote worker nodes. A stochastic tournament strategy approach is used for the selection and the replacement phases of the algorithm. Furthermore, in addition to classical simple diversification and intensification operators, *e.g.* random mutation, two-points crossover, each algorithm encloses an analogous set of conjugate-gradient extended operators. The defined alternate set of operators function by first applying the enclosed mutation, respectively, crossover standard mechanisms, the resulting offspring(s) being further refined by the local search component. Embedding the standard and the gradient enhanced version, a combined operator is provided, allowing for a selective, rate dependent, application of the internal sub-operators, *e.g.* allowing for the standard mutation operator to be applied on 90% of the subjected solutions, respectively, for the extended operator on the remaining 10%. An eloquent practical exemplification is found when considering a high-energy barrier surrounded optimum conformation. With no refinement, a close to optimum solution is subject to attain, with a high probability, an elevated fitness energy. Consequently, the solution, although encoding valuable information, exhibits a high probability of being discarded, in the selection process. A balanced design has to be assured, nevertheless, *e.g.* by specifying appropriate operator rates, gradient steps, etc., as to avoid a potential premature convergence of the algorithm. Additionally, a refined local optimum solution stands as a key minima representative over the surrounding high-energy conformations, locally characterizing the afferent landscape region.

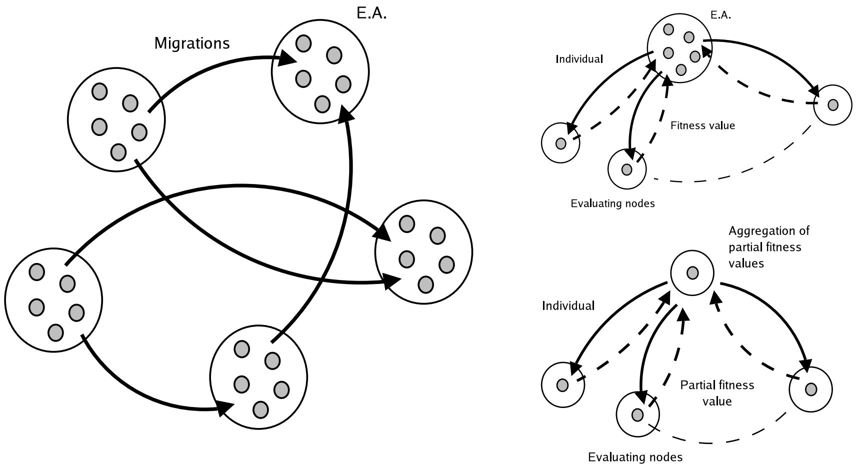


Fig. 5. The three main EAs parallelization models: island (a)synchronous cooperative model – left side of the figure, parallel evaluation of the population and distributed evaluation of a single solution – right side, upper, respectively lower part.

A synchronous ASA multi-start local search refinement phase is additionally interposed, succeeding the completion of a fixed number of iterations. Independent local explorations are simultaneously launched, for each of the to be refined solutions, obtained by random selection out of the local population. Further, allowing for convergence and diversity control, an asynchronous inter-islands exchange of genetic material is performed, at a predefined number of iterations. A cyclic, ring topology model communication pattern is set, *i.e.* accepting only one source and one destination per island. The specified migration model, allows for a coordinated global convergence, as determined by the migration frequency, number of exchanged solutions, etc., whilst reducing the external impact on the local island exploration process. A strong local attractor is required to cycle the entire ring, through multiple selection steps, before attaining global acceptance. Emigrant solutions are retrieved by means of a stochastic tournament selection, at the opposite end, the worst individuals in the target population being replaced by immigrant solutions. Survival of the best individual is assured by a weak-elitism scheme. For each local search refinement and migration phase, one tenth, respectively, one sixth of the population, is subject to undergo the local optimization, respectively, information exchange process.

Note that, except for selection and replacement, all operations, including the local search enhanced operators, are performed in parallel by delegation to worker nodes. A detailed discussion of the ParadisEO framework architecture and the afferent components developed in order to sustain the construction of the herein presented algorithmic model, execution roles, communication topologies, etc., is presented in [7, 8].

3.5 ParadisEO Based Implementation

ParadisEO[3], initially designed and developed by Sebastien Cahon [7, 8], is an extendible open source C++ framework based on a clear conceptual separation of the meta-heuristics from the problems they are intended to solve. The *EO* suffix stands for Evolving Objects, the framework being basically an extension of the Evolving Objects (EO) [33] LGPL C++ open source project, the result of an European joint work [33]. EO includes a paradigm-free Evolutionary Computation library, dedicated to the flexible design of EAs through evolving objects, superseding the most common dialects (Genetic Algorithms, Evolution Strategies, Evolutionary and Genetic Programming).

Furthermore, most common parallel/distributed models, *i.e.* synchronous island model, synchronous multi-start, etc., are provided in the ParadisEO-PEO module (Parallel EO). A portable design over distributed-memory machines and shared-memory multi-processors is offered, relying on standard libraries such as Message Passing Interface (MPI) [23, 24] and POSIX Threads (PThreads) [6]. A transparent exploitation of the enclosed parallel models, in (non) dedicated parallel environments, is assured. Nevertheless, with the continuous evolution of the distributed computing grids and with the perpetuous development of the available computing resources, there is a *sine qua non* requirement to pass beyond the physical design of the grids. Extending the existing framework, in order to offer a grid-enabled ParadisEO implementation, demands for a Grid middleware layer and a Grid Application Programming Interface. Furthermore, an infrastructure interface is required, providing communication and resource management tools. The here adopted approach consists in using the Globus Toolkit [20, 19] computing system as a Grid Infrastructure - an outline is presented in [52].

A layered architecture of the ParadisEO framework is presented in Fig. 6. From a top-down view, the first level supplies the optimization problems to be solved using the framework. The second level represents the ParadisEO framework, including optimization solvers, embedding single and multicriterion meta-heuristics (evolutionary algorithms and local searches). The third level provides interfaces for standard MPI based programming. At this level virtually any standard conforming MPI distribution may be placed as layer. The fourth and lowest level supplies communication and resource management services. A broad range of experimentations were conducted on employing the Globus Toolkit with MPICH/MPICH-G2 [23], MPICH-VMI [43] and OpenMPI [22].

With no exception, all tests have been deployed on the Grid'5000 (https://www.grid5000.fr) French nation-wide experimental computational grid, connecting several sites which host clusters of PCs interconnected by RENATER[4] (the French academic network). At this time, Grid'5000 is gathering more than 4000 computational cores with more than 100 Tb of non-volatile storage capacity, regrouping nine centers: Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes,

[3] http://paradiseo.gforge.inria.fr
[4] http://www.renater.fr

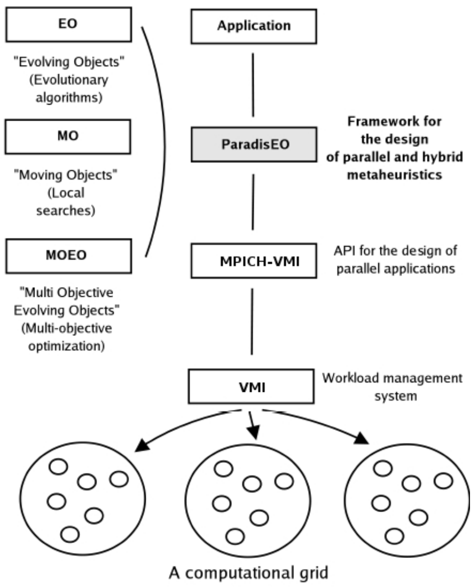

EO
"Evolving Objects" (Evolutionary algorithms)
MO
"Moving Objects" (Local searches)
MOEO
"Multi Objective Evolving Objects" (Multi-objective optimization)
Application
ParadisEO
Framework for the design of parallel and hybrid metaheuristics
MPICH-VMI
API for the design of parallel applications
VMI
Workload management system
A computational grid

Fig. 6. A layered architecture of ParadisEO.

Sophia-Antipolis, Toulouse. Following time dependent requirements and computational resources availability constraints, determined by the shared nature of the environment, experimentations were conducted on most of the Grid'5000 sites. As dictated by a per experiment demand, a varying number of resources has been used, ranging from a reduced number of computational cores, for tuning and prototyping purposes, to up to almost 1000 cores for the actual deployment and testing - see Table 1 for details.

Table 1. Environment details for a conformational sampling experimentation cumulating almost 1000 computational cores, over multiple clusters

Cluster/Site*	**CPUs**	**Cores**	**Architecture Details**
Azur/Sophia	59	118	Dual AMD Opteron^{TM} 2.0GHz/1MB/333MHz, 2GB RAM
Helios/Sophia	53	212	Quad Core AMD Opteron^{TM} 2.2GHz/1MB/400MHz, 4GB RAM
Sol/Sophia	27	108	Quad Core AMD Opteron^{TM} 2.6GHz/1MB/667MHz, 4GB RAM
Sagittaire/Lyon	60	120	Dual AMD Opteron^{TM} 2.0GHz/1MB/400MHz, 2GB RAM
Capricorne/Lyon	51	102	Dual AMD Opteron^{TM} 2.4GHz/1MB/400MHz, 2GB RAM
Orsay/Paris	152	304	Dual AMD Opteron^{TM} 2.4GHz/1MB/NA, 2GB RAM
Overall	**402**	**964**	

4 Experimental Outcomes

4.1 Conformational Sampling Benchmarks

Assessing conformational sampling algorithms requires to set a trade-off over the considered benchmarks. A first aspect to be considered regards complexity matters, *i.e.* reduced size conformations are of no interest (there is no need of determining the structure of a water molecule using computational grid resources) whilst highly complex molecules may be highly computationally restrictive (due to resource constraints, force field calibration limitations, etc.). A second aspect is defined on validation requirements - the crystallographic structure of the benchmark molecule has to be known in order to be able to have performance evaluations.

The herein adopted molecular complexes for the conformational sampling algorithms assessment, are the *tryptophan-cage* (trp-cage - Protein Data Bank ID: 1L2Y), the *tryptophan-zipper* (trp-zipper - Protein Data Bank ID: 1LE1) and the *α-cyclodextrin*. *Tryptophan-cage* and *tryptophan-zipper* belong to the class of mini-proteins presenting particularly fast folding characteristics. *Cyclodextrins*, in α, β or γ conformations, with 6, 7, 8 glucose units, respectively, due to their toroidal structure, are important for drug-stability applications, being used as protectors against micro-environment interactions or as homogeneous distribution stabilizers, etc.

The selected benchmark conformations can be considered, to a certain extent, as being significant and representative as they include different structural patterns, hence, requiring a flexible enough algorithm to predict the different enclosed secondary structures. Refer to Fig. 7 for a graphical representation of the three molecular conformations. An equivalent schematic representation is also exposed in order to better illustrate the structural characteristics of each molecule (as the cyclic structure of α-cyclodextrin). The α-cyclodextrin molecule, while not being a protein, has been included in the study due to its particular cyclic structure. In addition, the addressed conformations, given the number of defined torsional angles, namely 64, 54, 73 angles for α-cyclodextrin, 1LE1, 1L2Y, respectively, offer the advantage of not requiring an extremely expensive energy evaluation computation time.

4.2 Execution Configuration and Outcomes

A ring insular model consisting of three algorithms has been deployed at run-time, each island evolving a fixed-size population of 300 solutions for 300 generations. No specific parameter tuning has been considered, the employed configuration being incrementally constructed in a series of trial executions. As previously outlined, combined mutation and crossover operators have been employed, *e.g.* the classical two-point crossover operator and the conjugate-gradient enhanced version, in mutual exclusive manner, with a 0.85, respectively, a 0.15 rate. Analogously, the mutation operators are applied with equal rates, for the classical and local search extended version, having an overall 0.05 probability. A selection rate of 0.75 has been set, with a 0.95 probability of accepting a better individual over a worse one.

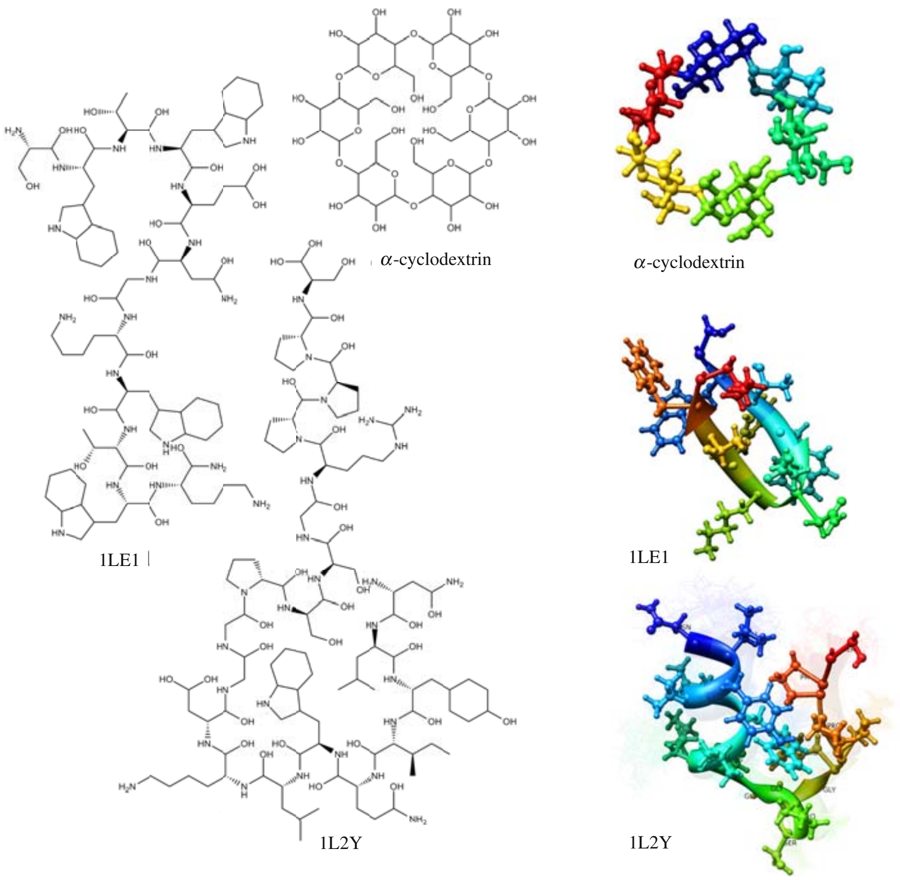


Fig. 7. Structural overview of the considered benchmarks – α-cyclodextrin, tryptophan-zipper (1LE1) and tryptophan-cage (1L2Y).

Although the induced fitness degradation, with a 0.05 probability, worse solutions are accepted in order to exploit the potentially significant enclosed information. Replacement is conducted on similar basis, with a 0.75 probability of discarding a worse solution. The refinement phase has been set to be applied at every five generations, relying exclusively on the ASA component, described in Section 3.3. A fine-grained gradient minimization is additionally carried out on the resulting conformations, exploiting the analytical foundations of the conjugate gradient local search operator. A worse-replacement strategy is used for reinserting the final refined solutions into the initial population.

Another element with important consequences over the convergence of the constructed algorithm is given by the asynchronous migration rate. Frequent migrations may result in a premature convergence while distant migrations fall at the opposite extreme - exploration conducted on distinct algorithms with independent evolution

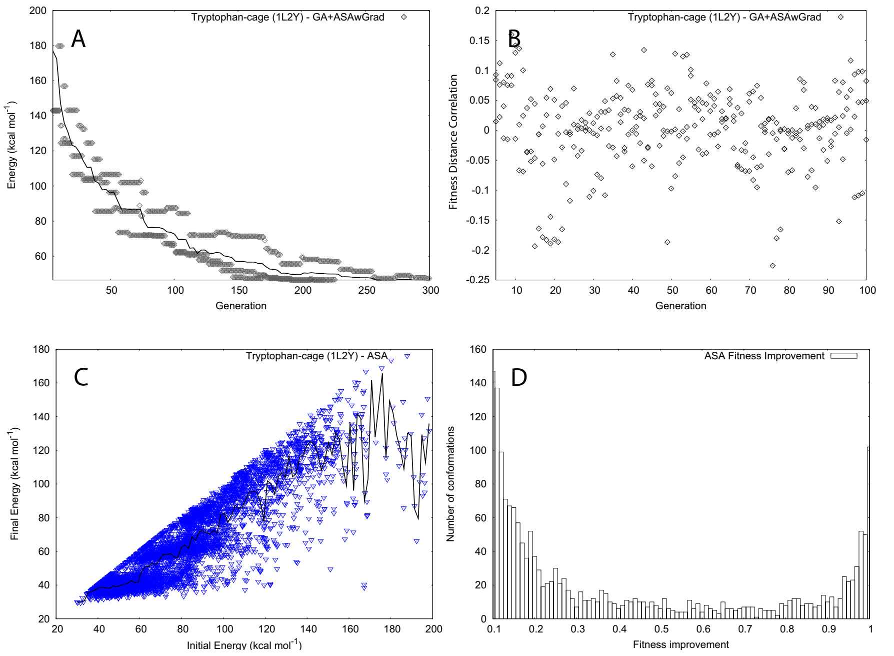


Fig. 8. 1L2Y – An execution example depicting the profile of the island model algorithm (A, B), in the second row, the evolution of the ASA component being captured (C, D).

curves. For the herein case, one sixth of the local population is set to emigrate, in asynchronous manner, at every five generations - migrations may occur at different times, depending on the advancement of each algorithm.

A meta-evolutionary genetic algorithm has been designed for finding an optimal parameterization of the adaptive simulated annealing algorithm. No special strategies or operators were designed, a simple distributed EA being considered; the algorithm has been executed inside the same grid environment. In this case each individual of the meta-algorithm represents an encoding of the different parameterization values – control parameters of the adaptive simulated annealing algorithm, initial temperature, number of accepted solutions determining reannealing, quenching factor, etc. The fitness of each individual has been computed as the average improvement obtained after running the adaptive SA on a set of five known difficult conformations. For each fitness evaluation run, for each of the five conformations, a maximum number of 3000 samplings was set. As an example, one of the chosen resulting parameterizations had a reannealing limit of 111 accepted conformations with 97 sampling points at each temperature and a large quenching factor of 33.16.

For the synchronous multi-start execution, two approaches were considered. The adaptive simulated annealing algorithm is either executed in order to sample 3000 solutions in one run, either 10 short runs with 300 samples each are iteratively launched. In addition, at the end of one ASA run, the outcome conformation is further optimized by applying a 30 step gradient.

As a first remark, after compiling the execution results, the use of conjugate gradient extended operators determined a dramatic improvement. Analyzing, for example, the results obtained by using the genetic algorithm alone, for the α-cyclodextrin conformation, an average of 3790.56 $kcal\ mol^{-1}$ (stdev. 708.54 $kcal\ mol^{-1}$) has been attained, with a maximum, minimum of 5845.27 $kcal\ mol^{-1}$, respectively 2470 $kcal\ mol^{-1}$. At the opposite extreme, the set of solutions found by the gradient hybridized genetic algorithm resulted in an average of 201.37 $kcal\ mol^{-1}$ (stdev. 21.82 $kcal\ mol^{-1}$), with a minimum of 161.69 $kcal\ mol^{-1}$ and a maximum of 243.05 $kcal\ mol^{-1}$. A number of 30 independent executions were performed for the gradient hybridized GA as well as for the GA alone, with no hybridization.

Finally, for all studied benchmarks, the ASA-hybridized GA (best scored conformations) attained a below native reference energy: 28.9 $kcal\ mol^{-1}$ for the 1L2Y protein (reference energy at 46.6 $kcal\ mol^{-1}$), -3.5 $kcal\ mol^{-1}$ for 1LE1 (11.1 $kcal\ mol^{-1}$) and 161.6 $kcal\ mol^{-1}$ for α-cyclodextrin (242.4 $kcal\ mol^{-1}$). Nevertheless, although descending below the energy of the native conformation, the corresponding RMSD (Root Mean Square Deviation) values were constantly outside acceptable limits, with minimum values close to or above 4Å.

A graphical illustration, capturing the island model algorithm evolution, is given in Fig. 8. The depicted examples outline, in a first step, results obtained for the hybrid island based algorithm, while the second part offers an overall perspective of the ASA execution-time improvement rate. For each island, at every generation, the fitness of the best found conformation is depicted (A), a median trend evolution line being traced. Although the algorithms advance at different rates, with several thresholds, convergence is attained near 300 generations. A corresponding fitness distance correlation (FDC) [32] plot is additionally illustrated (B), offering an overview of the fitness dynamics, *e.g.* convergence rate information, over generations fitness variance, etc. An ideal case would consist of a 1.0 FDC value, expressing a perfect correlation between fitness and inter-solutions distance values, while, at the opposite end, a -1.0 FDC value indicates a completely uncorrelated landscape, providing no useful information. A symmetrical spread may be observed (B), with an ascending positive correlation trend, as determined by the advancement of the exploration. Additionally, an outline of the ASA improvement bias is shown in the second row of the figure, traced as a plot exposing initial vs. final energy (C) and, second, as a histogram (D). Approximately one sixth of the refined conformations allowed for an above 10% improvement while only a reduced fraction of 3% resulted in an above 90% improvement (D). The equivalent run-time evolution graph (C), exclusively considering the ASA refinement outcomes, revealed several clusters, attributed to strong attractors determining basins in the conformational landscape (visible at $\sim$ 40.0 $kcal\ mol^{-1}$, 60.0 $kcal\ mol^{-1}$, final energy - C).

As an overall conclusion, first, the hybrid parallel algorithm design incurs strong exploration capabilities, although, second, far from native outcome conformation were returned. Appearing as energy landscape artifacts, with high RMSD - low energy conformations, due to the force field parameterization, the obtained solutions do not stand as valid conformations. As a consequence it can be concluded that a higher level extensive exploration approach is required with a more robust evaluation protocol.

4.3 Advanced Hybrid Hierarchical Parallel Algorithm

As determined by the drawn conclusions, a cluster sampling, domain decomposition oriented algorithm has been considered. A straightforward extension of the representation model has been constructed by considering, for a chromosome, an overlapping associated domain. Defining symmetric boundaries, for a given conformation, $\gamma = (\gamma_1, \gamma_2, \cdots, \gamma_N)$, a landscape domain is delimited, further denoted as $<\gamma,\ \eta> \equiv ([\gamma_1 - \eta_1, \gamma_1 + \eta_1],\ [\gamma_2 - \eta_2, \gamma_2 + \eta_2],\ \cdots,\ [\gamma_N - \eta_N, \gamma_N + \eta_N])$. The introduced definition and representation synthetically maps, over the conformer concept, nevertheless encompassing a less conformational structure significance, *i.e.* no underlying *specific* base template is associated to the given domain. Therefore, the term of *cell* is preferred in the following, describing, by direct association, a bounded structural subspace, as opposed to conformer, in order to designate a $<\gamma,\ \eta>$ entity. For simplicity, the assumption of having $\eta_i = \delta,\ 1 \leq i \leq N$, is considered in the following, where δ represents an *a priori* fixed arbitrary positive value. Additionally, having as basis the formulated assumption, a direct notation $<\gamma> \equiv <\gamma,\ \eta>$, with $\eta_i = \delta,\ 1 \leq i \leq N$, is employed in the following, as to designate a cell. An intuitive graphical representation is given in Fig. 9, depicting the transition from a highly multimodal energy landscape to a smoother, conformer fitness space. From an implementation point of view, the representation is constructed as an extension of the previously defined model, permitting the reuse of the entire developed algorithmic architecture, with no or less modifications.

A direct evaluation would consist of considering a $<\gamma>$ cell as designating an ensemble of solutions. Consequently, the problem resides in defining an appropriate evaluation function which, for a specified δ value and for a given cell, $<\gamma>$, offers a coherent evaluation, quantifying the *stability* of an overlapping conformer. Nevertheless, no complete characterization of a particular cell is possible, unless accounting for the cumulated interaction and contribution of an infinite number of conformations, confined within the cell boundaries. Consequently, an extrapolation formalism has to be defined, the evaluation function being constrained to infer on a *finite*, *representative* subset of conformations. Furthermore, the evaluation function has to be *reproducible*. Otherwise stated, assuming that representative independently sampled subsets $\mathscr{S}_i \subseteq <\gamma>,\ i \in \mathbb{N}$ are given, *comparable* evaluation results have to be provided. The construction of a *representative* cell subset hence demands for a within cell sampling to be performed, algorithmic basis being provided by the already defined approach.

Holding for the aforementioned specifications and having as support intuitive underlying physical concepts, a Gibbs free energy evaluation is considered – refer to Section 3.1 and Horvath *et al.* [26] for additional references and details. The function relies on the individual evaluation of a set of sampled solutions, $\gamma^s \in \mathscr{S}$,

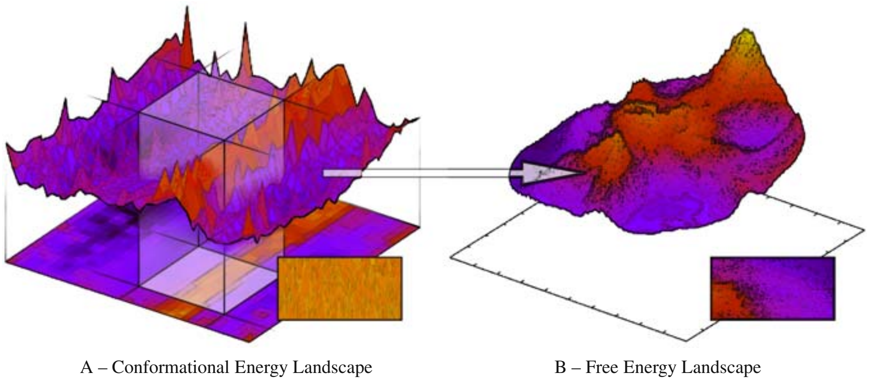

A – Conformational Energy Landscape
B – Free Energy Landscape

Fig. 9. Underlying conceptual basis of a free energy clustered sampling algorithm. A finite set of solutions, sampled within the boundaries of a specified cell (A), is employed for constructing a free energy evaluation – resulting in a singular free energy surface point (B). A more tractable landscape is obtained at the price of a higher computational load.

$\mathscr{S} \subseteq <\gamma>$. Extrapolating over the formalization details, an entropy equivalent measure is obtained, offering a characterization of the within $<\gamma>$ cell key minima depth and width. As determined by the nature of the evaluation function, a less sensitive to extreme perturbation energy values evaluation is attained, resulting in a smoothing effect. A graphical simplified corresponding exemplification, for the 1L2Y protein, is illustrated in Fig. 9.

As mentioned in the previous paragraphs, the construction of a *representative* set has to be addressed, as part of the fitness function definition. A first design decision consists in determining an optimal δ value. High values result in a reversion towards the initially addressed problem while, at the opposite end, reduced values imply the exploration space to be segmented into a large number of cells. The former case, while offering the advantage of simplifying the search space clustering, requires the support of a thorough intensive sampling, posing a reproducibility problem and, hence, inducing a high computational load. In analogous manner, the latter case, while assuring for *representative* sampled sets, results in an expensive exploration process, due to an explosion of the number of cells to be explored. With no or less information acquired, at the extreme case, the initial conformational energy landscape is potentially reproduced. Consequently, a sampling algorithm dependent balance has to be assured in order to allow for a pertinent segmentation of the search space and as to exploit the information which can be derived by assessing an ensemble of conformations. Therefore, a second correlated design decision, concerns the exploration algorithm to be employed – a random sampling would stand as a simple and fast candidate solution although offering no *reproducibility* guarantee, unless reduced size cells are considered. An exploration intensive approach, allowing for the search to be conducted over extended landscape domains, although enforcing the imposed demands, can potentially result in a redundant oversampling.

Assembling the introduced representation model and the free energy evaluation function, a meta-evolutionary algorithm has been constructed, the exploration being conducted over clusters in the conformational energy space. A hierarchical design is offered, comporting multiple parallelization levels. As highly complex aspects are addressed, no effective approach can be defined unless extensive distributed computational resources are employed. A first parallelization layer is inserted at the global meta-exploration level, the evaluation of each solution being synchronously delegated to local samplers. Further, each of the sampling processes deploys several island algorithms, for each island, a parallel evaluation of the conformations being performed at each generation, with additional synchronous multi-start refinement and migration processes. A schematic representation is given in Fig. 10. Note that, following the parallelization hierarchy, a highly scalable approach is attained, as determined by the decomposition of the parallel tasks. Given that, the implied design decisions mainly depend on the selected local sampling algorithm, the defined architecture is presented starting with the lower exploration layer as to end with the meta-exploration algorithm level.

As main criterion in proposing a local sampler solution, the requirement for an exploration intensive algorithm has been considered, as to allow for free energy evaluations on large cells within acceptable reproducibility limits. As demonstrated by the previous results, the algorithmic model proposed in Section 3.4, stands as a powerful candidate solution. Consequently, the same exact architecture has been used, with several modifications as detailed in the following. In depth details and analysis test cases, standing as basis for the herein obtained results, were also presented in [54, 55], addressing multiple operators, local search algorithms, adaptive and dynamic mechanisms, etc. Nonetheless, as we are here interested in exposing the hierarchical nature of the algorithms, opposing local and global sampling paradigms, no further details are here included.

Conducting several trial experimentations, it has been determined as coherent and sufficient to set a value of $\delta = 45$, corresponding to a $\pi/4$ angular value and allowing for wide extended cells to be defined. Further, a discrete representation has been adopted, where, for a $<\gamma>$ cell corresponding genotype, the enclosed $<\gamma_i>,\ 1 \leq i \leq N$, *loci* has been defined as having values from the $\{0, \cdots, 7\}$ set, with a corresponding angle value in the $[\delta(<\gamma_i> -1),\ \delta(<\gamma_i> +1)]$ interval. An inter-cells overlap has been allowed as to avoid boundary constraints, *e.g* torsional angle values requiring fine tuning near boundary limits. Note that the representation employed by the local sampling algorithm has not been modified, a mapping being defined as to assure the coherence of the representation.

Having as a pragmatic constraint the requirement of allowing for a fast sampling process to be conducted, a reduced population size of only 30 solutions has been assigned, for each island of the sampling algorithm, to be evolved over 10 generations. The exact same configuration of the operators and inter-algorithm migration topologies has been maintained, as presented in Section 3.4, with a down-scaling of the afferent parameter values. Local search refinement has been set to be triggered at every 5 generations, additionally, migrations being performed at every 2 generations, with an exchange of 10 individuals. Furthermore, a maximum of 10

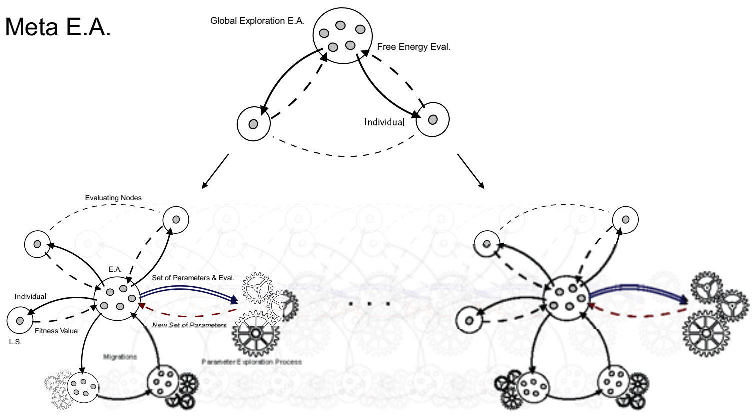


Fig. 10. A conceptual model depicting the architecture of the meta-exploration algorithm. Global exploration is carried at a conformer level, for each conformer associated cell, local sampling based free energy evaluations being computed.

conjugate gradient steps has been set, as opposed to the initial default configuration of 30 steps. As the exploration is carried out inside specified cell boundaries, all the determined solutions, as provided by each island, contribute to the construction of a representative sampling set. Therefore, at the end of the sampling process, a screening is performed, a set of the best found 30 distinct conformations being assembled. The gathered set further stands as basis for computing the free energy evaluation, characterizing the initial subjected cell.

Discrete combined operators have been employed, as to maintain a coherence of the representation, without introducing repairing mechanisms. Mutation has been defined as to be carried on a swap, random flip and a complete shuffle operators, with equal rates and with a 0.3 overall probability. In analogous manner, a uniform and a two-points crossover operators, with equal rates and with a 0.95 overall probability have been specified. A fitness sharing selection strategy is included, the distance, for two specified cells, $<\gamma^a>$, $<\gamma^b>$, being defined as the percentage of positional different *loci*. In this context, two solutions are considered to be part of the same cluster if found at a distance below 0.25, *i.e.* less than a quarter of the *loci* having different values. Additionally, the replacement is carried on a stochastic tournament strategy, with weak elitism enabled and with a 0.95 probability of discarding a worse solution over a better one.

At execution time, a maximum walltime of 50 hours has been imposed, the algorithm being executed in successive runs over a variable number of computational resources, with an average of ~400 cores. The algorithm has been set to evolve a population of 30 solutions for 100 generations, each solution defining a cell to be sampled. As resulting from the obtained outcomes, the proposed approach offered impressive results – refer to Fig. 11 for a graphical illustration. As an example, for the 1LE1 protein, the algorithm ranked first the cell centered around the native reference, within the cell, the first ranked solution, with a -13.32 $kcal\ mol^{-1}$

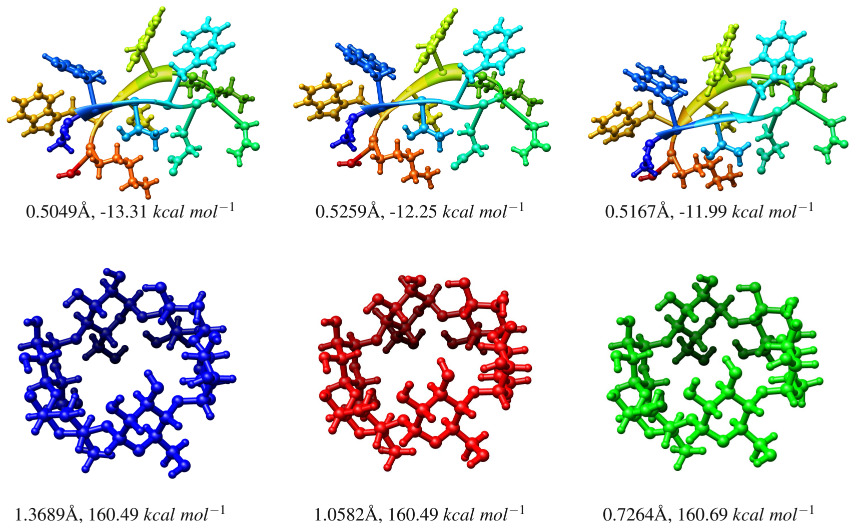

0.5049Å, -13.31 *kcal mol*$^{-1}$
0.5259Å, -12.25 *kcal mol*$^{-1}$
0.5167Å, -11.99 *kcal mol*$^{-1}$
1.3689Å, 160.49 *kcal mol*$^{-1}$
1.0582Å, 160.49 *kcal mol*$^{-1}$
0.7264Å, 160.69 *kcal mol*$^{-1}$

Fig. 11. Tryptophan-zipper (first row) and α-cyclodextrin (second row) – best found conformations, ranked in concordance with the associated energy.

conformational energy fitness, standing as a perfect match, with a 0.5049Å RMSD. Additionally, an average RMSD of 0.6431Å has been attained for the 30 first ranked conformations, with a minimum, maximum RMSD of 0.3611Å (-9.23 *kcal mol*$^{-1}$), respectively 2.0860Å (-7.14 *kcal mol*$^{-1}$). In similar manner, for the α-cyclodextrin molecule, for the top 30 ranked conformations, a 3.7595Å average has been attained, with a minimum, maximum value of 0.5313Å (162.01 *kcal mol*$^{-1}$), respectively 8.9869Å (675.54 *kcal mol*$^{-1}$) – remarkable to notice, only 4 out of the 15 first ranked conformations had an RMSD above 1.0Å. As exposed in Fig. 11, for the first three α-cyclodextrin conformations, an RMSD of 1.3689Å (160.49 *kcal mol*$^{-1}$), 1.0582Å (160.49 *kcal mol*$^{-1}$), respectively 0.7264Å (160.69 *kcal mol*$^{-1}$) has been obtained. Although no similar results have been attained, in the given time frame, for the *tryptophan-cage* protein, undergoing independent studies, carried out in the context of the Docking@Grid project, confirmed an over-fitting bias of the employed force field, resulting in non-consistent results when addressing α-helices vs. β-sheets patterns.

5 Conclusions and Future Work

Allowing for extreme hybrid constructions to be defined and enclosing intrinsic parallel support, evolutionary algorithms comport, nevertheless, a high structural complexity level. Different evolutionary parallel models were employed, initial experimentations standing as a proof for the intensive exploration capabilities of the approach. An extension of the initial approach was defined, addressing conformers instead of singular conformations. A free energy evaluation function was introduced in the model, endorsing the evaluation of clusters of conformations as an ensemble and quantifying the width and the depth of the representative conformer minima region. Impressive results were attained for the *tryptophan-zipper* protein and for the α-cyclodextrin conformational benchmark, with a below 1.0Å RMSD average for the first 30 ranked 1LE1 conformations. All experimentations were conducted on Grid'5000 [11], different MPI distributions [23, 43, 22] being employed at execution time. To conclude with, an effective high-performance parallel hybrid conformational sampling algorithm was constructed, answering the initially defined *ANR Dock Project – Conformational Sampling and Docking on Computational Grids* directions.

An unlimited number of consequent prospective directions may be considered, enforcing the obtained outcomes, the exploration of novel parallel paradigms, etc. A consequential study entailing exploration approach enhancements stands as an adjacent objective in order to encompass high-throughput conformational screening support. Finally, extensive background for arising technologies, *e.g.* General Purpose GPUs (Graphics Processing Units) [5], MPICH-G4/MPIg, etc., is considered as to expand the ParadisEO framework, including fault-tolerance, desktop computing and volatile environments support.

References

1. Alba, E., Talbi, E.G., Luque, G., Melab, N.: Metaheuristics and parallelism. In: Parallel Metaheuristics: A New Class of Algorithms. Wiley Series on Parallel and Distributed Computing, vol. 4, pp. 79–104. Wiley, Chichester (2005)
2. Alder, B., Wainwright, T.: Phase transition for a hard sphere system. Journal of Chemical Physics 27, 1208–1209 (1957)
3. Alder, B., Wainwright, T.: Studies in molecular dynamics. i. general method. Journal of Chemical Physics 31, 459–466 (1959)
4. Bernstein, F., Koetzle, T., Williams, G., Meyer Jr., E.F., Brice, M., Rodgers, J., Kennard, O., Shimanouchi, T., Tasumi, M.: The protein data bank: a computer-based archival file for macromolecular structures. Journal of Molecular Biology 112, 535–542 (1977)
5. Buck, I.: Gpu computing: Programming a massively parallel processor. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO 2007), p. 17. IEEE Computer Society, Washington (2007)
6. Butenhof, D.: Programming with POSIX Threads. Professional Computing Series. Addison-Wesley Longman Publishing Co., Boston (1997)
7. Cahon, S., Melab, N., Talbi, E.G.: Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. Journal of Heuristics 10(3), 357–380 (2004)
8. Cahon, S., Melab, N., Talbi, E.G.: An enabling framework for parallel optimization on the computational grid. In: Proceedings of International Symposium on Cluster Computing and the Grid (CCGrid 2005), vol. 2, pp. 702–709. IEEE Computer Society, Washington (2005)
9. Calland, P.Y.: On the structural complexity of a protein. Protein Engineering 16(2), 79–86 (2003), http://peds.oxfordjournals.org/cgi/content/abstract/16/2/79

10. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell (2000)
11. Cappello, F., Caron, E., Dayde, M., Desprez, F., Jegou, Y., Primet, P., Jeannot, E., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Quetier, B., Richard, O.: Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed. In: Proceedings of IEEE/ACM International Workshop on Grid Computing (GRID 2005), pp. 99–106. IEEE Computer Society, Washington (2005), http://dx.doi.org/10.1109/GRID.2005.1542730
12. Cozzone, A.: Proteins: Fundamental chemical properties. Encyclopedia of Life Sciences, 1–10 (2002), http://doi.wiley.com/10.1038/npg.els.0001330
13. Crescenzi, P., Goldman, D., Papadimitriou, C., Piccolboni, A., Yannakakis, M.: On the complexity of protein folding. Journal of computational biology 5(3), 423–465 (1998)
14. Dauber-Osguthorpe, P., Roberts, V., Osguthorpe, D., Wolff, J., Genest, M., Hagler, A.: Structure and energetics of ligand binding to proteins: Escherichia coli dihydrofolate reductase-trimethoprim, a drug-receptor system. Proteins: Structure, Function, and Genetics 4(1), 31–47 (1988), http://dx.doi.org/10.1002/prot.340040106
15. Dill, K.: Theory for the folding and stability of globular proteins. Biochemistry 24(6), 1501–1509 (1985), http://view.ncbi.nlm.nih.gov/pubmed/3986190
16. Dorsett, H., White, A.: Overview of molecular modelling and ab initio molecular orbital methods suitable for use with energetic materials. Tech. Rep. DSTO-GD-0253, Department of Defense, Weapons Systems Division, Aeronautical and Maritime Research Laboratory, Salisbury, South Australia (2000)
17. Fletcher, R., Powell, M.: A rapidly convergent descent method for minimization. Computer Journal 6, 163–168 (1963)
18. Fletcher, R., Reeves, C.: Function minimization by conjugate gradients. Computer Journal 7, 149–154 (1964)
19. Foster, I., Kesselman, C.: The Grid: Blueprint for a new computing infrastructure. Morgan Kaufmann Publishers, Los Altos (2003)
20. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of High Performance Computing Applications 15(3), 200–222 (2001), http://dx.doi.org/10.1177/109434200101500302
21. Garwin, L., Lincoln, T.: A Century of Nature: Twenty-One Discoveries that Changed Science and the World. University of Chicago Press, Chicago (2003)
22. Graham, R., Shipman, G., Barrett, B., Castain, R., Bosilca, G., Lumsdaine, A.: Open mpi: A high-performance, heterogeneous mpi. In: Proceedings of CLUSTER. IEEE, Los Alamitos (2006), http://dblp.uni-trier.de/db/conf/cluster/cluster2006.html#GrahamSBCBL06
23. Gropp, W.: Mpich2: A new start for mpi implementations. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J., Volkert, J. (eds.) PVM/MPI 2002. LNCS, vol. 2474, p. 7. Springer, Heidelberg (2002)
24. Gropp, W., Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI: The Complete Reference, vol. 2. MIT Press, Cambridge (1998)
25. Hestenes, M., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards 49(6), 409–436 (1952)
26. Horvath, D., Brillet, L., Roy, S., Conilleau, S., Tantar, A.A., Boisson, J.C., Melab, N., Talbi, E.G.: Local vs. global search strategies in evolutionary grid-based conformational sampling & docking. In: Proceedings of IEEE Congres on Evolutionary Computation, CEC 2009 (2009)

27. Ingber, L.: Adaptive simulated annealing (asa), global optimization c-code. Tech. rep., Caltech Alumni Association (1993)
28. Ingber, L.: Simulated annealing: Practice versus theory. Journal of Mathematical Computation Modelling 18(11), 29–57 (1993)
29. Ingber, L.: Adaptive simulated annealing (asa): Lessons learned. Control and Cybernetics 25, 33–54 (1996)
30. Ingber, L.: Adaptive simulated annealing (asa) and path-integral (pathint) algorithms: Generic tools for complex systems. Tech. rep., Chicago, IL (2001)
31. Ingber, L., Rosen, B.: Genetic algorithms and very fast simulated reannealing: A comparison. Mathematical Computer Modeling 16(11), 87–100 (1992)
32. Jones, T., Forrest, S.: Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In: Proceedings of International Conference on Genetic Algorithms, pp. 184–192. Morgan Kaufmann, San Francisco (1995)
33. Keijzer, M., Guervós, J., Romero, G., Schoenauer, M.: Evolving objects: A general purpose evolutionary computation library. In: Proceedings of European Conference on Artificial Evolution (EA 2002), pp. 231–244. Springer, London (2002)
34. Kirkpatrick, S., Gelatt, C., Vecchi, M.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983), citeseer.ist.psu.edu/kirkpatrick83optimization.html
35. Krauter, K., Buyya, R., Maheswaran, M.: A taxonomy and survey of grid resource management systems for distributed computing. Software Practice and Experience 32(2), 135–164 (2002), citeseer.ist.psu.edu/krauter01taxonomy.html
36. Lander, E.S., Linton, L.M., Birren, B., et al.: Initial sequencing and analysis of the human genome. Nature 409(6822), 860–921 (2001)
37. Little, P.: Dna sequencing: the silent revolution. In: A Century of Nature: Twenty-One Discoveries that Changed Science and the World, ch. 16. University of Chicago Press, Chicago (2003)
38. McCammon, J.A., Gelin, B.R., Karplus, M.: Dynamics of folded proteins. Nature 267(5612), 585–590 (1977)
39. Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E.: Equation of state calculations by fast computing machines. The Journal of Chemical Physics 21(6), 1087–1092 (1953), http://link.aip.org/link/?JCP/21/1087/1
40. Neumaier, A.: Molecular modeling of proteins and mathematical prediction of protein structure. SIAM Review 39(3), 407–460 (1997), citeseer.ist.psu.edu/neumaier97molecular.html
41. Ngo, J.T., Marks, J.: Computational complexity of a problem in molecular structure prediction. Protein Engineering 5(4), 313–321 (1992), http://peds.oxfordjournals.org/cgi/content/abstract/5/4/313
42. Pande, V., Baker, I., Chapman, J., Elmer, S., Khaliq, S., Larson, S., Rhee, Y., Shirts, M., Snow, C., Sorin, E., Zagrovic, B.: Atomistic protein folding simulations on the submillisecond timescale using worldwide distributed computing. Biopolymers 68(1), 91–109 (2003)
43. Pant, A., Jafri, H.: Communicating efficiently on cluster based grids with mpich-vmi. In: Proceedings of CLUSTER, pp. 23–33. IEEE Computer Society, Los Alamitos (2004), http://dblp.uni-trier.de/db/conf/cluster/cluster2004.html#PantJ04
44. Polak, E., Ribière, G.: Note sur la convergence des méthodes de directions conjuguées. Revue française d'informatique et de recherche opérationnelle 16, 35–43 (1969)
45. Ponder, J., Case, D.: Force fields for protein simulations. Advances in Protein Chemistry 66, 27–85 (2003)

46. Rabow, A., Scheraga, H.: Improved genetic algorithm for the protein folding problem by use of a Cartesian combination operator. Protein Science 5(9), 1800–1815 (1996), http://www.proteinscience.org/cgi/content/abstract/5/9/1800
47. Schulten, K., Phillips, J., Kale, L., Bhatele, A.: Biomolecular modeling in the era of petascale computing. In: Bader, D. (ed.) Petascale Computing: Algorithms and Applications, pp. 165–181. Chapman & Hall/CRC Press, Boca Raton (2008)
48. Shewchuk, J.: An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA (1994), http://portal.acm.org/citation.cfm?id=865018
49. Shirts, M., Pande, V.: COMPUTING: Screen Savers of the World Unite! Science 290(5498), 1903–1904 (2000)
50. Stewart, C., Müller, M., Lingwall, M.: Progress towards petascale applications in biology: Status in 2006. In: Proceedings of Euro-Par Workshops, pp. 289–303 (2006)
51. Talbi, E.G.: A taxonomy of hybrid metaheuristics. Journal of Heuristics 8(5), 541–564 (2002)
52. Tantar, A.A., Melab, N., Demarey, C., Talbi, E.G.: Building a virtual globus grid in a reconfigurable environment - a case study: Grid5000. Tech. Rep. inria-00168130, INRIA, France (2007), http://hal.inria.fr/inria-00168130/en
53. Tantar, A.A., Melab, N., Talbi, E.G.: A grid-based genetic algorithm combined with an adaptive simulated annealing for protein structure prediction. Soft Computing 12(12), 1185–1198 (2008)
54. Tantar, A.-A., Melab, N., Talbi, E.-G.: An analysis of dynamic mutation operators for conformational sampling. In: Biologically-inspired Optimisation Methods: Parallel Algorithms, Systems and Applications. Studies in Computational Intelligence. Springer, Heidelberg (2009)
55. Tantar, E., Tantar, A.A., Melab, N., Talbi, E.G.: Analysis of local search algorithms for conformational sampling. In: Advances in Parallel Computing, Parallel Programming and Applications on Grids, P2P and Networked-based Systems. IOS Press, Amsterdam (2009)
56. Venter, J., Venter, J., Adams, M., et al.: The sequence of the human genome. Science 291(5507), 1304–1351 (2001), http://www.sciencemag.org/cgi/content/abstract/291/5507/1304
57. Van de Waterbeemd, H., Carter, R., Grassy, G., Kubinyi, H., Martin, Y., Tute, M., Willett, P.: Glossary of terms used in computational drug design. Pure and Applied Chemistry 69(5), 1137–1152 (1997)
58. White, A., Zerilli, F., Jones, H.: Ab initio calculation of intermolecular potential parameters for gaseous decomposition products of energetic materials. Tech. Rep. DSTO-TR-1016, Department of Defense, Energetic Materials Research and Technology Department, Naval Surface Warfare Center, DSTO-TR-1016, Melbourne Victoria 3001, Australia (2000)

Laser Dynamics Modelling and Simulation: An Application of Dynamic Load Balancing of Parallel Cellular Automata

J.L. Guisado, F. Jiménez-Morales, J.M. Guerra, F. Fernández de Vega, K.A. Iskra, P.M.A. Sloot, and Daniel Lombraña González

1 Introduction

This chapter reviews the application of a biologically inspired heuristic technique – Cellular Automata (CA) – for developing high performance simulations of a well known complex system: the laser.

J.L. Guisado
Departamento de Arquitectura y Tecnología de Computadores, Universidad de Sevilla.
E.T.S. Ingeniería Informática. Avda. Reina Mercedes s/n. 41012 Sevilla, Spain
e-mail: jlguisado@us.es

F. Jiménez-Morales
Departamento de Física de la Materia Condensada, Universidad de Sevilla.
P.O. Box 1065, 41080 Sevilla, Spain
e-mail: jimenez@us.es

J.M. Guerra
Departamento de Óptica, Facultad de CC. Físicas, Universidad Complutense de Madrid.
28040 Madrid, Spain
e-mail: jmguerra@fis.ucm.es

F. Fernández de Vega
Centro Universitario de Mérida, Universidad de Extremadura.
Sta. Teresa Jornet, 38. 06800 Mérida (Badajoz), Spain
e-mail: fcofdez@unex.es

K.A. Iskra
Argonne National Laboratory, Mathematics and Computer Science Division.
9700 South Cass Avenue, Argonne, IL 60439, USA
e-mail: iskra@mcs.anl.gov

P.M.A. Sloot
Section Computational Science. Laboratory for Computing, System Architecture and Programming. Faculty of Science, University of Amsterdam.
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
e-mail: p.m.a.sloot@uva.nl

D. Lombraña González
Centro Universitario de Mérida, Universidad de Extremadura.
Sta. Teresa Jornet, 38. 06800 Mérida (Badajoz), Spain
e-mail: daniellg@unex.es

CA can be described as a class of mathematical systems. They were introduced several decades ago, and are well suited to model spatio-temporal phenomena. On the other hand, CA can be implemented very efficiently on parallel platforms, given both, their intrinsic parallel nature, with all the components working usually in a synchronized way, and the discreteness of the individual components using the same behavior rules. We therefore make use of this feature, and consider the problem of running Parallel CA simulations on non-dedicated clusters of workstations. We thus present results of laser dynamics simulations, traditionally modeled using differential equations.

This new approach can be very useful when modeling lasers given that differential equations are difficult to integrate or even difficult to apply: lasers ruled by stiff differential equations, devices with complex boundary conditions, very small devices for which the approximations implied by the differential equations may not be valid, etc.

The presented model is based on a synchronous CA using the Single Program, Multiple Data (SPMD) paradigm, deployed on a non-dedicated cluster of computers. Therefore, it is not clear in advance if a good performance and efficiency can be obtained on this kind of non-dedicated platform. We thus analyze the feasibility of executing our parallel bioinspired model of laser dynamics on an heterogeneous non-dedicated cluster, and we evaluate its performance including artificial load to simulate other tasks or jobs submitted by other users. Finally, a dynamic load balancing strategy is used with two main differences from previous CA implementations:

- It is possible to migrate the load to cluster nodes that do not belong initially to the pool.
- The model uses the load balancing tool –the Dynamite system– to give flexibility to the model.

By studying the performance and scalability of this parallel implementation we obtain very satisfactory results, including performance increases from 60% to 80%.

This chapter is organized as follows: In Section 2, we will present the problem to be solved by the proposed algorithm: laser dynamics. We will describe in detail the proposed algorithm in Section 3. In Section 4, we will review some of the laser properties which are successfully reproduced by the CA model. Next, we will describe a parallel implementation of the CA model and we will analyze its performance and scalability when executed on a heterogeneous non-dedicated cluster, including dynamic load balancing. In Section 6, we will propose some ideas for future work. Finally, some conclusions will be drawn in Section 7.

2 The Problem: Laser Dynamics

A laser is a device that generates and amplifies coherent electromagnetic radiation based on the stimulated emission phenomenon, predicted by Albert Einstein in 1917 [1]. The word laser is an abbreviation of "Light Amplification by Stimulated Emission of Radiation". In a laser system radiation is amplified by propagation across a medium, in which, the population of an upper energy state is larger than the population of a low energy state (population inversion). Some mechanism, usually known as the pumping system, is needed to enhance the upper state population up to be larger than the remaining in a lower energy state. When the pumping is above a threshold value, the radiation traveling through the medium is amplified by the stimulated emission process. The effective amplification is usually enhanced by placing the laser active medium inside a Fabry-Perot resonator, that provides a feed-back by making the amplified light bounce between two parallel mirrors. Therefore, the laser device acts as a regenerative light oscillator and transient, periodic or chaotic oscillatory processes can be originated in it.

The time dependence of the total number of laser photons and the total population inversion in the laser medium can be described [2], as a first step, by the Equations (1) and (2):

$$\frac{dn(t)}{dt} = KN(t)n(t) - \frac{n(t)}{\tau_c} \tag{1}$$

$$\frac{dN(t)}{dt} = R - \frac{N(t)}{\tau_a} - KN(t)n(t) \tag{2}$$

This model, based on two coupled nonlinear rate equations is simplified but can still describe realistically many laser dynamics phenomena. The first equation provides the variation on the number of laser photons $n(t)$ with time, proportional to the laser beam intensity. The term $+KN(t)n(t)$ describes the increase in the number of photons by stimulated emission (K is the coupling constant between the radiation and the population inversion). The term $-n(t)/\tau_c$ accounts for the decaying (or absorption) process of laser photons inside the laser cavity with a characteristic decay time τ_c. The second equation represents the temporal variation of the population inversion $N(t)$. The term $+R(t)$ represents the pumping of electrons with a pumping rate R to the upper laser level. The term $-N(t)/\tau_a$ introduces the decaying of electrons from the upper laser level to lower levels, with a characteristic decay time τ_a. The product term $-KN(t)n(t)$ reflects the decreasing of the population inversion by stimulated emission. The presence of the product term $KN(t)n(t)$ in each equation gives them a nonlinear nature. For small amplitude fluctuations its solutions can show relaxation oscillations in their evolution towards a steady state. For strong oscillations the two variables $n(t)$ and $N(t)$ are changing in a fast and typically nonlinear way and there does not seem to be a simple analytic solution [2, 3].

The four-level laser system shown in Figure 1 is a simplified model that still gives a realistic description of the main phenomena featured by a laser system: for instance, an external pumping process can excite electrons and make them jump from the ground level up to level E_3. Similarly, the figure shows the population inversion process, produced between levels E_1 and E_2 thanks to the fact that the life times of energy levels E_3 and E_1 are negligible as compared to the life time of level E_2. Therefore, electrons in levels E_3 and E_1 decay very fast but level E_2 is metastable. On the other hand stimulated emission occurs when an electron in

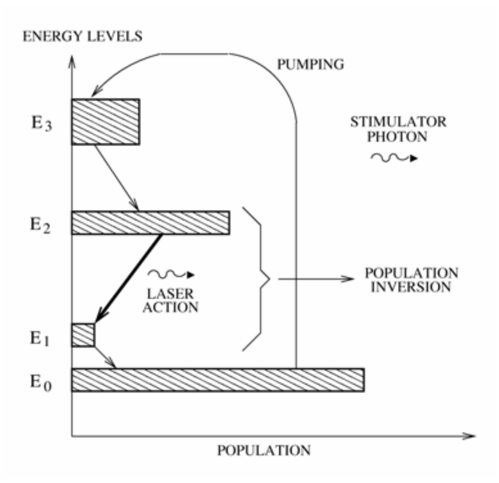


Fig. 1. A four-level laser system and its basic physical processes

level E_2 decays down to level E_1 stimulated by the presence of a stimulator photon with energy $E = E_2 - E_1$. In addition, there are two processes which are also very important and which are not represented in Figure 1: absorption of electrons in level E_2 (which decay to lower levels due to different processes not related to stimulated emission) and absorption of laser photons, a fraction of which disappear because they leave the laser cavity through the semi-reflecting mirror or are absorbed by the material.

3 Cellular Automata and Laser Dynamics

Cellular Automata (CA) are a class of spatially and temporally discrete mathematical systems characterized by local interaction and synchronous dynamical evolution [4]. They provide an excellent approach for modeling and simulating complex systems and have been used over the recent years in many fields of science and technology [5, 6]. We study here the modeling of light amplification by stimulated emission by means of CA, firstly described by Guisado et. al. in [3].

The algorithm is based on a two-dimensional, partially probabilistic, multi-variable CA that simulates a transverse section of the active medium in a laser system. The defining characteristics of the CA are described in the following.

3.1 Cellular Space

The CA employ a cellular space consisting of a two-dimensional square lattice which contains $N_c = L \times L$ cells. Periodic boundary conditions are used.

3.2 State of the Cells

Each of the cells within the CA embodies two variables: $a_{ij}(t)$ and $c_{ij}(t)$. The first one, $a_{ij}(t)$, represents the state of the electron in cell $\{ij\}$ (row i and column j) at time t: when $a_{ij}(t) = 0$ the electron is in the ground state and when $a_{ij}(t) = 1$ the electron is in the upper laser state. Also, $c_{ij}(t) \in \{0, 1, 2, ..., M\}$ represents the number of laser photons in cell $\{ij\}$ at time t. This number is bounded by an upper value M which must be chosen large enough to avoid the saturation of the system. The state variables represent "bunches" of real photons and electrons. Their values are linked to the number of photons and electrons in the real system by a normalization constant.

3.3 Neighborhood

Every cell performs local interactions with a predefined number of surrounding cells. We employ the well-known *Moore neighborhood* for establishing interactions patterns: the neighborhood of any particular cell is formed by the cell itself (C), its four nearest neighbors located at the north (N), south (S), east (E) and west (W) positions and the four next nearest neighbors located at the northeast (NE), southeast (SE), southwest (SW) and northwest (NW) positions, as shown in Figure 2.

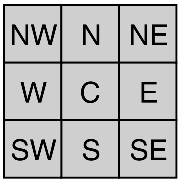


Fig. 2. Moore neighborhood

3.4 Transition Rules

Every CA model requires a set of rules which defines the behavior and evolution of the whole system. This set of rules, usually known as transition rules, specify the state of each cell at time step $t + 1$ depending on its state and the state of the cells included in its neighborhood at time step t. Therefore, the rules model the physical processes working at the microscopic level in the laser system. The application of the transition rules is the main operation of a CA algorithm. In our case the overall structure of the CA laser model algorithm is shown in Algorithm 1.

Algorithm 1. Pseudo-code diagram for the CA laser model

```
Initialize system
Input data
for time step = 1 to maximum time step do
    for each cell in the array do
        Apply stimulated emission rule
        Apply photon decay, electron decay, and pumping rules
        Apply noise photons creation rule
    end for
    Calculate populations after this time step
    Optional additional calculations on intermediate results
end for
Final calculations
Output results
```

After an initialization step, the transition rules are applied synchronously to each CA cell inside a time loop. Our CA model employs five transition rules:

- Stimulated emission rule: If the electronic state of a cell has a value of $a_{ij}(t) = 1$ at time t and the sum of the values of the laser photons states in the nine neighboring cells is larger than a certain threshold θ (which in our simulations has been taken to be 1), then at time $t+1$ a new photon will be emitted in that cell: $c_{ij}(t+1) = c_{ij}(t) + 1$ and the electron will decay to the ground level: $a_{ij}(t+1) = 0$. All the cells of the CA must be updated in parallel. To this end, changes from this rule are computed using a temporal matrix c'_{ij}. After the rule has been applied to all the cells of the CA, the values of c_{ij} are updated with the contents of c'_{ij}.
- Photon decay rule: Each photon is destroyed τ_c time steps after being created. In particular, (tlc_{ijk}) represents the number of time steps that will have to elapse until a particular photon located in cell $\{ij\}$ (at row i and column j) is destroyed, where k distinguishes between the different photons that can occupy the same cell. When a photon is created, $tlc_{ijk} = \tau_c$. After that, 1 is subtracted from tlc_{ijk} at each time step and the photon will be destroyed when $tlc_{ijk} = 0$.
- Electron decay rule: After an electron is excited from the ground level to the upper laser level, it will decay to the ground level again after τ_a time steps, if it has not yet decayed by stimulated emission. In particular, (tla_{ij}) represents the number of time steps that will have to elapse until a particular electron located in cell $\{ij\}$ decays to the ground level. When the electron is initially excited, $tla_{ij} = \tau_a$. After that, 1 is subtracted from tla_{ij} at each time step and the electron will decay to the ground level again when $tla_{ij} = 0$.
- Pumping rule: If the electronic state of a cell $\{ij\}$ has a value of $a_{ij}(t) = 0$ at time t, then at time $t+1$ that state will have a value of $a_{ij}(t+1) = 1$ with a probability pumping λ.
- Noise photons creation: A small number of laser photons in randomly chosen positions is introduced at each time step to reproduce spontaneous emission and thermal contributions, responsible of the initial laser start-up. To this end, for

a small number of randomly chosen cells $\{ij\}$ ($< 0.01\%$ of total) it is applied $c_{ij}(t+1) = c_{ij}(t) + 1$.

This CA models a typical four levels laser system, as described in section 2 and represented by the diagram shown in Figure 1. Thus it has been assumed that the population of the lower laser level (level 1 in Figure 1) is negligible. For this reason, stimulated absorption transitions have not been taken into account. Also for the same reason, the state $a_{ij}(t) = 0$ doesn't correspond to the lower laser level, but to the ground level, which is coupled to the pumping mechanism.

4 Experimental Analysis

We first present a summary of experimental results obtained when applying the previously described model to the simulation of laser dynamics (for interested readers, a whole description of results can be found in [3, 7, 8, 9]). We analyze in this section how the CA model of laser dynamics can reproduce different aspects of the phenomenology of laser systems.

Three main parameters influence the behavior of the system: the pumping probability (λ), the life time of photons (τ_c) and the life time of excited electrons (τ_a). In a simulation, an initial state is provided ($a_{ij}(0) = 0$, $c_{ij}(0) = 0$, $\forall ij$, except a small fraction, 0.01%, of noise photons present) and then the system is let to evolve for a number of time steps. In each step, we measure two macroscopic magnitudes: the total number of laser photons, $n(t)$, and the total number of electrons in the upper laser state (population inversion), $N(t)$, defined in Equation 3.

$$n(t) = \sum_{i=1}^{L_x} \sum_{j=1}^{L_y} \mathrm{c_{ij}}(t), \qquad N(t) = \sum_{i=1}^{L_x} \sum_{j=1}^{L_y} a_{\mathrm{ij}}(t) \tag{3}$$

It is a well-known laser systems' feature that laser action only happens when the pumping probability is over a threshold value. This property is correctly reproduced by the CA model [3] and the dependence of this threshold value on the other two system parameters (life times τ_a and τ_c) is found to be in good agreement with the laser behavior, as shown in Figure 3.

Two different behaviors which depend on the values of their three main parameters can be found in their time evolution: a constant or an oscillatory behavior [3, 8]. As shown in Figure 4, the model reproduces these two kinds of behavior: the time evolution obtained from the simulations is similar to that one exhibited by laser systems, described for example in [2]. A lattice size of 400×400 cells was used for this figure.

Moreover, we can also notice in Figure 5 another complex behavior in the CA model: irregular oscillations with fluctuations on a wide range of time scales appear (see [8]). This regime could correspond to a chaotic state, as found in the dynamics of many lasers. Also, the dependence on the system parameters of the type of

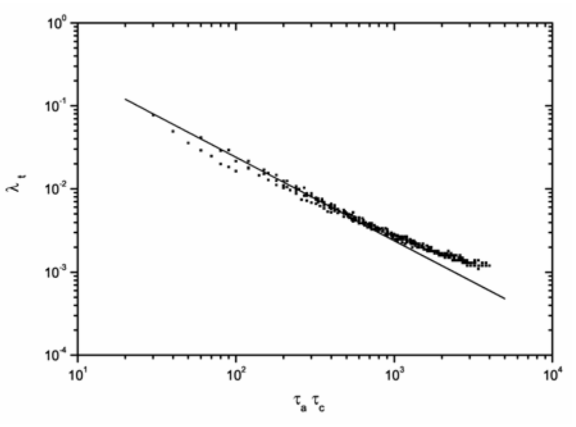


Fig. 3. Dependence of the threshold pumping probability λ_t from the CA laser model on the product of the characteristic life times τ_a and τ_c (measured in time steps), plotted on a logarithmic scale. The solid line is the laser behavior predicted by the standard laser rate equations, and the dots are the results of the simulations.

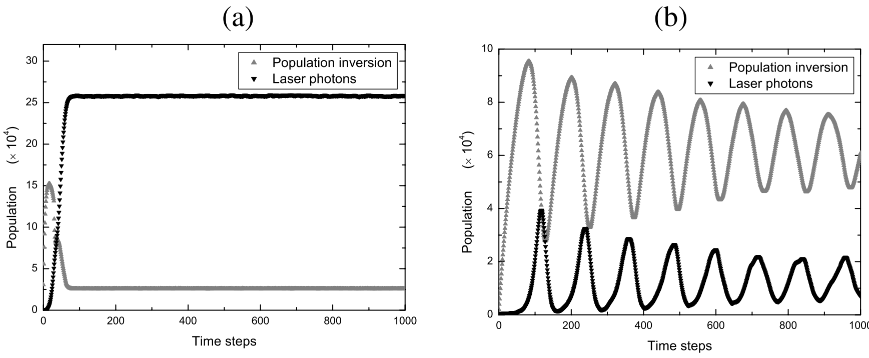


Fig. 4. Results of the simulations, showing the time evolution of the two macroscopic magnitudes —number of laser photons $n(t)$ and population inversion $N(t)$— versus time, for two different sets of values of the system parameters. The two main characteristic behaviors exhibited by lasers are reproduced by the CA model: (a) (left): Constant behavior. Parameters: $\{\lambda = 0.192,\ \tau_c = 10,\ \tau_a = 30\}$. (b) (right): Oscillatory behavior. Parameters: $\{\lambda = 0.0125,\ \tau_c = 10,\ \tau_a = 180\}$.

behavior exhibited in the time evolution of the system is in a good qualitative agreement with the laser behavior [3], as shown in Figure 6.

In this figure, we show a Contour plot of a magnitude called the *Shannon's entropy* of the distribution of the number of laser photons, for a fixed value of $\tau_c = 10$ time steps and obtained using simulations with a 200×200 lattice. This magnitude is a good indicator of the presence of oscillations in the time evolution of the number of laser photons (for a precise definition and discussion, see for example [7]).

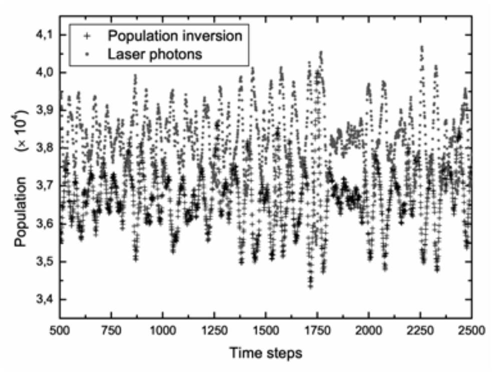


Fig. 5. Regime with irregular oscillations, for: $\lambda = 0.031$, $\tau_c = 10$, $\tau_a = 180$. The number of laser photons and population inversion are plotted versus time, after a transient of 500 time steps. Lattice size: 400×400 cells.

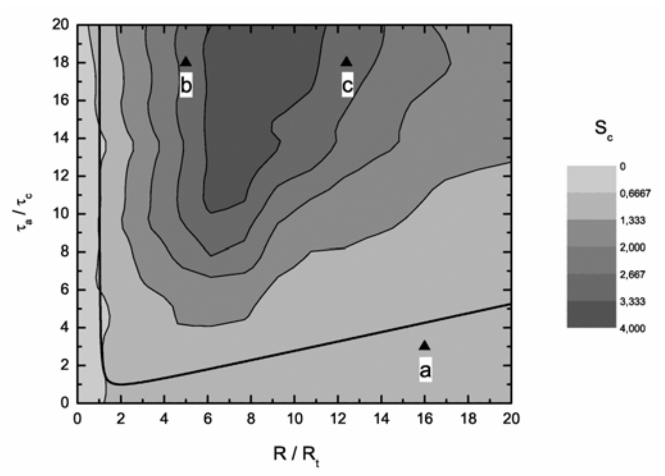


Fig. 6. Contour plot of the Shannon's entropy of the distribution of the number of laser photons obtained from the simulations with a fixed value of $\tau_c = 10$ time steps. This plot shows there is a good qualitative agreement between the dependence on the system parameters of the type of behavior exhibited by the system, as obtained from the simulations, and the laser behavior, delimited by the black line.

In this plot, R is the laser pumping rate and R_t is the threshold laser pumping rate, which are linearly related to the pumping probability λ and the threshold pumping probability λ_t that appear in the CA model, so that $\frac{R}{R_t} = \frac{\lambda}{\lambda_t}$. Points *a*, *b* and *c* show the values of the parameters that correspond to Figs. 4 and 5: *a* corresponds to a constant behavior (Figure 4 left), *b* to a oscillatory behavior (Figure 4 right), and *c* to a regime with irregular oscillations (Figure 5).

High values of the Shannon's entropy (dark zones) correspond to an oscillatory behavior and low values (bright zones) to a non-oscillatory response. The predictions of the standard laser rate equations are indicated by the black line: areas of oscillatory behavior should appear above and to the right of this curve and constant behavior should appear in the remaining areas. There is a good qualitative agreement between the predictions and the results of the simulations indicated by the Shannon's entropy, as the high values of this magnitude appear above and to the right of the black line and their contour resemble the shape of this line.

5 Parallel CA Based Simulation of the Laser

As shown in previous section, the CA based laser model correctly reproduces much of the phenomenology of the laser system, and can be therefore considered an alternative to the standard modeling approach, which employs differential equations. Even when such a very simple coarse-grained CA model has demonstrated its usefulness, if we pursue more realistic simulations for specific laser devices, and we want a larger granularity, closer to real macroscopic systems, a 3D CA – or huge 2D CA instead– may be required. Therefore, a very large lattice size will be needed, which will make necessary parallel computers systems to run the model, avoiding thus the otherwise prohibitively large runtime of sequential counterparts.

In this section, we describe the parallel implementation of the previous CA model and study its performance and scalability running on a small computer cluster (interested readers can refer to [9], [10], [11] and [12] for a larger description of results obtained). We begin by reviewing previous approaches to parallel implementations of CA models.

5.1 Previous Approaches

As described above, sequential CA-based simulations can only be used for very simple systems. In order to simulate real world phenomena (which need 3D or large 2D CA) parallel implementations running on high performance parallel computers must be used since very long computing time or memory requirements are needed [13].

During the last decade some attempts to introduce parallelism within CA have been described. Most of them were not intended to implement in a direct way the inherently parallel CA internal rules of working, which can be easily simulated in a sequential fashion, but to improve speedup of the whole process by using a large number of processors.

The first attempts to parallelize CA were carried out by M. Resnick with the StarLogo system [14] and by Cannataro et. al. [15], although many approaches and results have been described later by using parallel CA, such as CAMEL [16], Nemo [17], PECANS [18], DEVS [19] and P-CAM [20]. A review of the topic is presented by Talia in [13].

Two main kinds of hardware infrastructure can be found in the literature for implementing parallel CA. The first one consists of using parallel computers. The second one requires specialized hardware such as the Cellular Automata Machine (CAM) [21]. In this work, we focus on using available parallel, cluster or GRID deployments. In fact, general-purpose parallel computers are well-suited for scalable CA models, from the point of view of speedup, programmability and portability.

Considering the structure of CA, the parallel implementation must take care of the information about the state of the cells included in the borders of the different partitions of the system; this information must be exchanged after each time step, as represented in Fig. 7. Therefore some methodology is required for the implementation of the parallel CA that allows information exchange among the different processes involved. Two main solutions are available: using general purpose parallel programming languages, such as HPF, HPC++ or Linda, or employing a standard high-level sequential language combined with specific libraries allowing parallel applications to run, such as MPI (Message Passing Interface), PVM (Parallel Virtual Machine) or OpenMP (Open Multi Processing).

On the other hand, when considering the information that must be exchanged among the different processes implementing the parallel CA after each time step, all of them must wait until all the computing nodes have finished for each time step before proceeding, i.e. the system operates in a lock-step mode. Therefore, the performance of the parallel implementation is limited by the slowest running task. A group of overloaded nodes which execute the computations slower than the majority of the nodes can degrade the overall performance. As the usual platform for executing this kind of applications are non-dedicated (and often heterogeneous) clusters, it raises the following question: Can this algorithm have a reasonably good performance when running on such platforms?

Several proposals in the literature focus on distributing the active cells between the nodes for CA featuring some cells that may become idle for a number of time steps [22, 15, 23]. Similarly, the possibility of moving cells from heavily loaded nodes to more unloaded ones has been described in [24, 25, 26]. A different possibility is to adjust the size of the partitions to be handled by each cluster node (see for instance [27, 28]). The idea behind those proposals is to make a balance on the

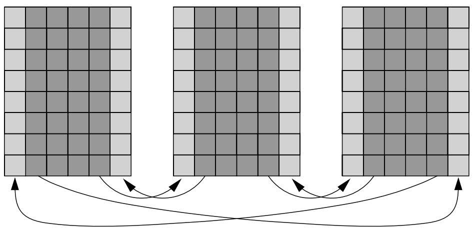


Fig. 7. In the parallel implementation of a CA, information of the state of the cells included in the borders of each partition of the system has to be communicated to the neighboring partition to be used in the computation corresponding to the next time step. In this example, the CA has been partitioned into parallel stripes. Each partition is assigned to a different processing node.

load when some nodes are overloaded. Yet, all of these proposals lack the capability of migrating jobs to new nodes which are dynamically added to the pool of computers initially running the CA model. This possibility, if present, would provide extra flexibility for a real-life non-dedicated parallel computing environment. This is the main reason why we opted for using a dynamic load balancing approach.

Even when load balancing has already been considered for parallel CA, most of the approaches include the dynamic load balancing mechanism within the CA algorithm (check for instance the P-CAM system [20] which directly integrates data decomposition and dynamic load balancing into the framework functionality). Our proposal tries to split and differentiate both important but non-related sections of the algorithm, the parallel CA and the load balancing technique. This would allow in the future to transparently change or improve any of them without affecting each other.

The tool that we have employed is *Dynamite* [29], an automated load balancing system that can migrate individual tasks which are part of a parallel program running with a message passing library. Dynamite is based on Dynamic PVM [30], a re-implementation of the PVM message passing library that includes the load balancing functionality. It monitors the utilization of the cluster nodes and migrates tasks when some of them get under-utilized or over-utilized as defined by configurable thresholds.

The Dynamite system is composed of three separate parts (see [31] and [32] for a complete description): the load-monitoring subsystem, the scheduler –which determines when a migration becomes necessary, which tasks will be involved and which particular allocation will be adopted– and the task migration software.

We have chosen Dynamite because of its flexibility, maturity, and availability. Nevertheless, dynamite is not the only load balancing system available. Other alternatives could also be used to execute this kind of simulations, such as the CAMELot-Grid system [33], and also the general purpose framework designed by Vadhiyar and Dongarra, implemented and tested in the GrADS system [34]. Even when they have some advantages over Dynamite, such as their possibility of integration on a grid computing environment, we preferred Dynamite. The main reason was our interest for running the experiments on a cluster computing environment –mainly because of the tightly coupled nature of the parallel CA model– : a parallel CA (a *high performance computing* application) requires low latency of the communications, and this cannot be generally attained on a GRID environment (which generally would be more adequate for running multiple executions of a complete CA for different values of the parameters –a *high throughput computing* application–).

5.2 Algorithm Description: Basic Approach

In order to parallelize the CA model, and taking into account all the considerations included above, we decided to employ the message passing paradigm. Given the selection of Dynamite as a component for implementing and analyzing dynamic load balancing mechanisms, we selected the *parallel virtual machine (PVM)* implementation of this paradigm. A master-worker model was implemented, such as the one described in Figure 8.

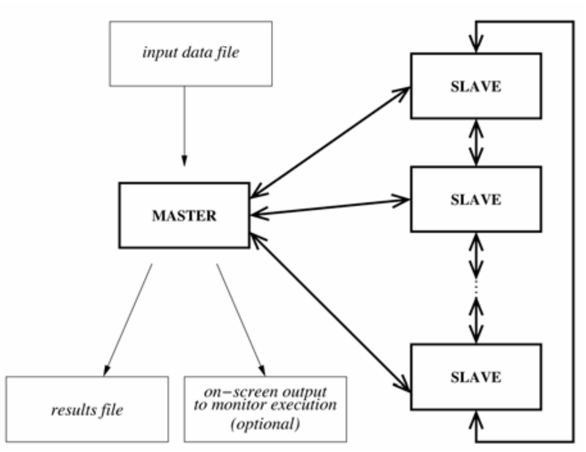


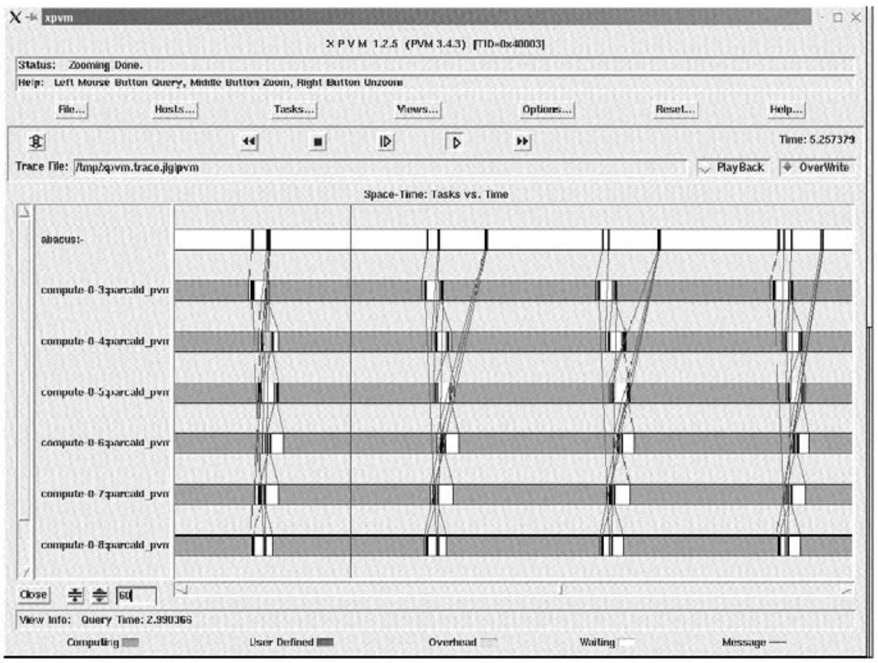
Fig. 8. Block diagram of the parallel implementation of the CA model of laser dynamics, showing processes running on different processors (boxes in bold type represent different processors), communications between them (bold lines) and data flows.

The *data decomposition methodology* is employed for distributing identical tasks with different partitions –of equal size– of the data among the pool of computers employed for the simulation, one partition per processor. Master and worker programs are therefore in charge of the following tasks:

- Master program:
  1. Read input data (system size, number of partitions, parameter values, number of time steps) and initialize.
  2. Spawning of slave programs.
  3. Partitioning of the initial data of the automaton.
  4. Sending of common information and initial data to each slave.
  5. Collection of results from slaves at each time step.
  6. Termination of slave programs.
  7. Calculations performed using collected data.
  8. Output of final data to external files.
  9. Timing functions to measure performance.
- Slave program:
  1. Reception of common information and initial data from master.
  2. Time evolution computation for the assigned partition: application of CA evolution rules.
  3. Exchange of state of the boundary cells with slave programs computing the neighboring partitions.
  4. Computation of intermediate results and their communication to master program.

Among the possibilities for establishing the domain decomposition (see [9]), a 1D decomposition has been used, so that the CA is divided into parallel vertical stripes, each of them assigned to a different computer. Extra ghost cells have been included both at the left and right sides of each partition (see Fig. 7) which are in charge of storing the state of border cells belonging to adjacent partition, and allows to compute all the transition rules for the cells. The state information required from neighboring cells is the photon state $c_{ij}(t)$, which will be sent from neighboring subdomains and stored in the ghost cells. Each slave program is responsible for computing the time evolution on its assigned partition.

At the beginning of each iteration the state of the boundary cells is directly exchanged between slave programs computing neighboring partitions, using two couples of PVM send and receive routines (`pvm_send` and `pvm_recv`). The routine `pvm_recv` is blocking, so it waits until the specified message has arrived. Therefore, this exchange plays the role of a synchronization point between all the slave programs. This is illustrated in Fig. 9 which shows a detail of the tasks executed by each node and the messages transferred between different nodes versus time, once the computation has started. This figure also shows that computation periods are



Status: Zooming Done.
Help: Left Mouse Button Query, Middle Button Zoom, Right Button Unzoom
Time: 5.257379
Trace File: /tmp/xpvm.trace.jlg|pvm
View Info: Query Time: 2.990366

Fig. 9. Gantt chart depicting a detail of the tasks executed by each cluster node and the messages transferred between different nodes versus time, once the calculation has started. The exchange of neighboring states between nodes processing adjacent partitions at the beginning of each iteration acts as a synchronization point.

much longer than communication periods, so that the application achieves a high computation-to-communication ratio, of the order of 10.

5.3 *Performance and Scalability Analysis*

We have measured the performance and scalability of the parallel CA by running simulations on the Beowulf-type cluster "Abacus" from the University of Extremadura (Spain), see Table 1.

Table 1. Abacus hardware specifications

Nodes	10
Microprocessor	Pentium-4
Clock Frequency	2.7 GHz (6 nodes) and 1.8 GHz (4 nodes)
Network	100 Mbps Fast-Ethernet switch

To avoid indeterminism in the results due to the heterogeneity of the cluster, for simulations with nodes 1 to 6, slave programs have always been run on the "fast" (2.7 GHz) machines, and for simulations with 7 to 10 nodes additional "slow" (1.8 GHz) machines have been used to complete the required number of nodes. The master program has always been run on the master node of the cluster (1.8 GHz).

Three different system sizes and different number of partitions –one per worker processor– have been considered for the experiments. Figure 10 shows results, and a significant runtime decrease can be noticed when the number of processors increase. The only exception is the change from 6 to 7 processors where an increase is registered due to the assigning strategy that has been used: only "fast" nodes are assigned to jobs with 6 or less processors and for jobs with more than 6 processors some "slow" processors have to be used.

In order to measure the performance of a parallel application, *speedup (S_p)* can be employed as defined by Foster in [35]. This value establishes a comparison between a parallel algorithm and its sequential counterpart. It can be defined as the ratio of the runtime of the sequential version of the program running on 1 processor of the parallel computer (T_1) to the runtime of the parallel version running on m processors of the same computer (T_m):

$$S_p(m) = \frac{T_1}{T_m} \tag{4}$$

Fig. 11 shows the speedup obtained for the parallel implementation of our CA model for the three different system sizes, compared to the *linear speedup* –line $y = x$. For the smallest system size, a very good performance has been obtained. For the other two system sizes, still better performance figures are obtained, in fact *super-linear speedup* (speedup higher than linear). The main reason is the finite memory space available for only one processor and therefore the necessity of using the swap memory. Because of this circumstance, the calculation of very large system

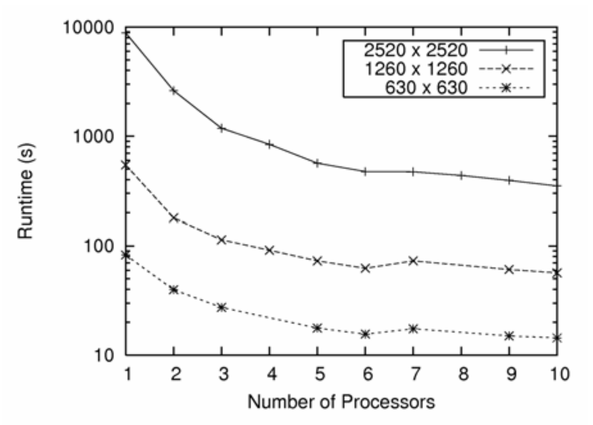


Fig. 10. Runtime of the experiments, using a logarithmic scale, for different number of partitions of the whole CA, each running on a different processor. Measurements for three different system sizes are shown.

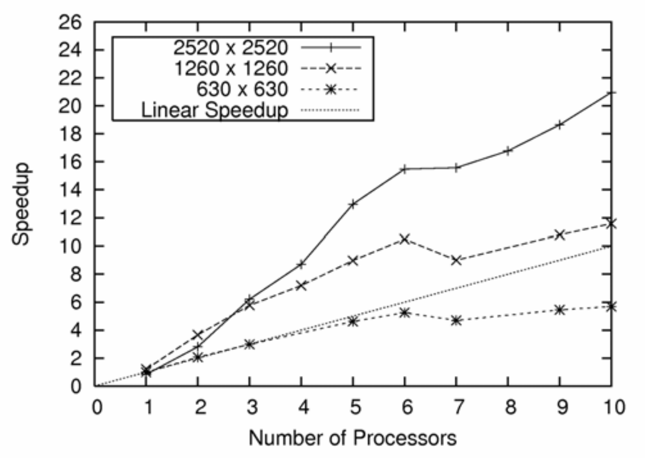


Fig. 11. Speedup obtained for the parallel implementation with respect to the sequential program for different number of processors and for three different system sizes. For comparison, the ideally optimal linear speedup has been shown. A very good performance is obtained for a moderate system size (630 $\times$ 630 cells) and a super-linear speedup for larger system sizes.

sizes –as for example a detailed 3D simulation– may not be affordable on a single PC (for the prohibitively large runtime needed due to the use of swap memory) but feasible on a cluster, in which the system is partitioned so each individual node needs less memory and does not have to use swap memory (interested readers can also check [9]).

In order to analyze the scalability of the combination parallel application-parallel computer, the running times obtained for the same experiment when increasing the system size (which in our problem is represented by the total number of cells of the CA) and the number of processors by the same factor have been compared. The results are shown in Figure 12. The same experiment as for Fig. 4(b) has been used, but involving the computation of 10000 time steps. For an ideally scalable application, the same running time should be obtained [36]. Our parallel application shows only a small excess (from 2 % to 5 %) of runtime compared to the optimal value. Therefore its scalability is good on a small computer cluster.

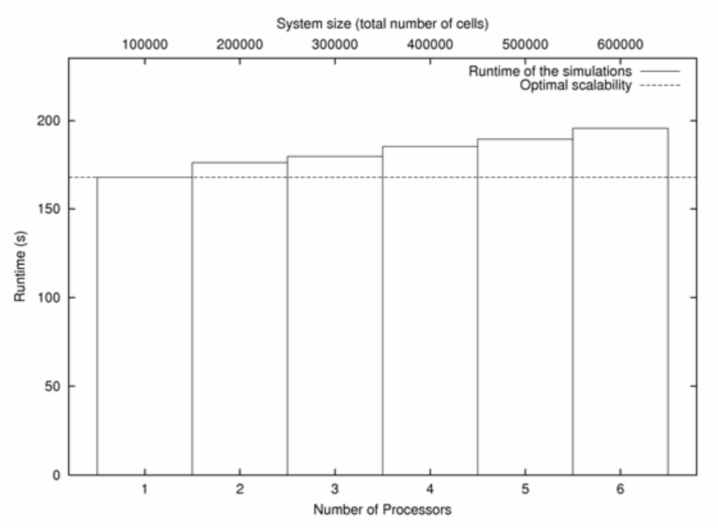


Fig. 12. Scalability analysis of the combination parallel application-parallel computer. The runtime for the same experiment but increasing the system size and the number of processors by the same factor is shown. The optimal ideal value would be the same runtime for all cases (horizontal line). The results show only a small excess (from 2 % to 5 %) with respect to this optimal value.

5.4 Load Balancing with Dynamite

Previous sections have shown the interest of deploying a parallel version of the CA based laser model. We consider now the problem of load balancing: how to obtain the best results when running experiments on a non-dedicated platform, where different tasks dynamically arrive to run simultaneously with our parallel application. The problem is thus to decide when and where different tasks must be migrated for improving speedup.

In order make this study we have included artificial loads that simulate a normal non-dedicated cluster use. All the experiments shown below have thus been run under controlled conditions on the cluster. We have considered the computation of the time evolution of the system during 10,000 time steps for a single value of the system parameters: $\lambda = 0.0125$, $\tau_c = 10$, $\tau_a = 180$.

A sequential program with a loop statement including a single assign instruction involving double precision numbers was employed as the external load, similarly as described in [34]. This C program was compiled regularly with no application of optimization techniques that would allow to improve the runtime of the program. The artificial load was intended to simulate the normal use of a non-dedicated high performance computing cluster for different users. Normally, to achieve the best performance possible, a cluster user would not run more than one process of her application on any cluster node. For that reason, only one artificial load process was executed on each cluster node.

The experiments used 6 worker nodes plus the master one for the parallel CA application, while 10 nodes were available on the cluster. The external load was systematically assigned to a number of cluster nodes, and time was then measured. The idea was to study the effect of different levels of loads for both the regular PVM version of the algorithm, and also the one employing Dynamite, which includes the load balancing system.

Immediately after starting the CA application, the artificial load task was initiated on a number of nodes, which range from 0 to 5 nodes and were always nodes to which one of the slave CA applications had been initially allocated also. The artificial load tasks kept on running for a time longer than the total execution time of the CA application.

5.4.1 Results and Discussion

Table 2 presents results obtained for the experiments described above, including execution time and improvement obtained when the load balancing technique is employed.

Table 2. Execution time and improvement due to load balancing when the application is run with and without load balancing and running artificial external load on a different number of cluster nodes. Normal PVM was used for configurations without load balancing and the Dynamite system for configurations with load balancing.

Configuration	Execution time (s)	Improvement
No load balancing with artificial load	1895.08	-
Load balancing with load on 1 node	384.59	80 %
Load balancing with load on 2 nodes	564.76	70 %
Load balancing with load on 3 nodes	611.12	68 %
Load balancing with load on 4 nodes	1595.75	16 %
Load balancing with load on 5 nodes	1833.82	3 %
No load, with and without load balancing	233.43	-

The first row shows the execution time obtained when no load-balancing technique is employed –standard PVM– while external loads are applied to any number of nodes, from 1 to 5. Regardless of the number of nodes undergoing external loads, the execution time is always the same. The reason is that the CA laser model operates in a lock-step mode, and the slowest running task limits the global performance obtained.

The following rows show the execution time when Dynamite is employed – instead of regular PVM, so that the load balancing mechanism is enabled. External artificial loads are applied again to nodes ranging from 1 to 5. Additionally, each row offers information about the improvement obtained when compared to previous execution –first row, when no load-balancing technique was employed.

Finally, the execution time obtained when running the application without any artificial load, which is the same with and without load balancing, has been shown as a reference in the last row.

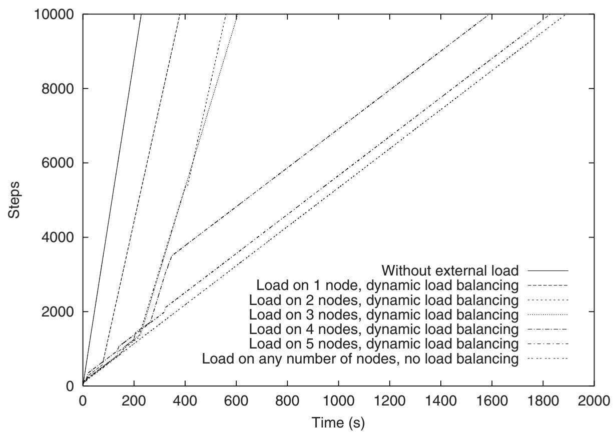


Fig. 13. Execution progress of the CA laser model application for different levels of artificial external load on the system. The system size was 840 × 840 cells. The number of cluster nodes used on the execution is 6.

Fig. 13 represents the number of executed time steps from the application versus time, which offers an idea of the progression of the application. We can notice an interesting improvement in the performance obtained when the parallel application is run on 6 worker nodes plus the master one, while a total number of 10 nodes are available on the cluster: when external load is run on up to 3 nodes, and given that idle nodes are available, the load balancing mechanism takes a good advantage, migrating required tasks, therefore reducing the execution time by a factor or 3. If more than 3 nodes receive external load, the improvement obtained is lower, but we still obtain a smaller execution time when compared to the first row of the table. The execution progress initially follows the same straight line as for no dynamic load balancing (i.e. for standard PVM), until the load balancing system identifies the situation and performs the migration of some of the tasks of the system to balance the load.

After that, when external loads are run on a small number of nodes, a significant improvement is found in the execution progress, following again a new straight line close to that one of the standard PVM. When considering external loads on a higher number of nodes, sometimes the benefits obtained after migrations produced by the load balancing mechanism are very low.

We have also found that occasionally, after an advantageous migration of tasks, the dynamic load balancing system incorrectly migrates tasks to let the system load unbalanced and obtain a sub-optimal execution progress.

It is also of interest to point out that the dynamic load balancing system incurs in practically no overhead on the execution time of the application, as its execution progress is virtually identical for PVM and Dynamite when there is no external load applied: the same line in Fig. 13 (labeled as ”Without external load”) applies to both cases.

We have also studied the effect of the system size on the global performance obtained. To this end, we have run simulations for three different system sizes and the execution progress has been compared.

The results of the experiments performed are shown in Fig. 14. Given that large CA sizes might require the use of swap memory of the operating systems, which would greatly decrease the whole performance of the system, keeping us from correctly analyzing runtime, we have employed relatively small CA sizes (interested readers can also see [11]). The figure shows that the use of a load balancing strategy results in a good performance improvement for all system sizes within the studied ranges.

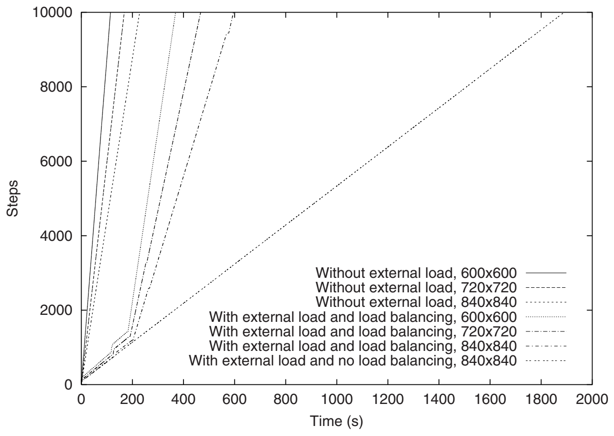


Fig. 14. Execution progress of the CA laser model application for different system sizes. The number of cluster nodes used on the execution is 6 and artificial external load has been run on 3 nodes.

Finally, we have studied the frequency and regularity of activation of the scheduling mechanism, so that a given experiment has been performed a number of times under exactly the same initial conditions.

Fig. 15 shows four different runs of the same experiment using Dynamite. We can notice some cases in which the load balancing system lets the load unbalanced and the execution time is not optimal. Although this behavior was also present in the experiments reported in previous figures for a 10% - 20% of the executions, these cases were not taken into account for the results presented.

We can thus conclude that migrations are not performed by the load balancing system in a very regular and deterministic way, and although results obtained are globally of interest , we clearly see that the scheduler component of the Dynamite load balancing system could be improved.

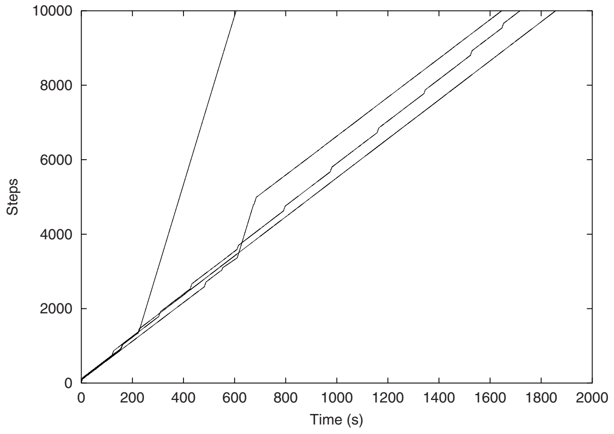


Fig. 15. Execution progress of 4 different runs of the application with Dynamite carried out under the same conditions. System size: 840×840 cells.

6 Future Work: Virtualization and Load Balancing

In the previous sections, we have reviewed the application of cellular automata to model laser dynamics and we have shown that with this approach it is possible to develop high performance simulations that run efficiently on computer clusters. We have also shown that these simulations can successfully be run on heterogeneous non-dedicated clusters, using an adequate load balancing mechanism, with a good performance. Finally, in this section we will present some ideas about future work in this subject, regarding the joint use of a dynamic load balancing tool like Dynamite and virtualization technology, to have a self-adapting cluster computing environment capable of deploying additional cluster nodes on demand, in the course of a computation.

Nowadays, the virtualization technology is gaining more adepts quickly. The benefits of using it as a solution for deploying and administering different services like

HTTP, FTP, etc. within the virtual machines, have achieved a great success. Big processor companies like Intel or AMD have their own and specific solutions to provide a good performance within the virtualization paradigm.

Virtualization is a technology that allows to multiplex the physical hardware to take advantage of its computer resources [37, 38, 39]. This technology creates a "virtual" machine (VM) where it is possible to load and run an operating system and its associated applications, for example a cluster node.

The VM is handled by a software called the *virtual machine monitor* (VMM) or hypervisor. The VMM is in charge of virtualizing the underlying hardware and assures that a problem or a bug within the VM will affect only to the VM and not to the real hardware and OS. Therefore, security and isolation are two of the main benefits of using VMs.

The main features of virtualization are the following:

- **Resource isolation.** Virtualization isolates each VM inside the host machine. This is very useful from the standpoint of the researcher because a failure inside a VM will affect only that VM and not the real machine.
- **Guest OS instantiation.** This feature permits to create an OS image that can be loaded into any machine that is compatible with the VMM employed for creating it.
- **Snapshots** or state serialization (also known as checkpointing [40]). With virtualization it is possible to freeze the execution of a whole OS and restart it exactly where it was stopped.

There are two different approaches to provide virtualization in x86 platforms. One of them is the *native virtualization*. The native virtualization implements this technology by providing an exact copy of the underlying hardware for the VM, in terms of functionality. This approach is two-folded: any OS is supported without modifications, but the performance gets affected due to x86 was not designed to be virtualized [41]. For this reason, any problematic instruction has to be captured by the VMM to assure the right operation of the virtual machine.

The other approach is the *paravirtualization*. This technique implements virtualization by providing a virtual hardware that is similar, instead of identical, to the underlying hardware in order to circumvent the previous described x86 problem. In order to use this technique, the guest OS has to be adapted to support the paravirtualization while applications can be run without any modifications (the binary interface is not modified).

Both techniques have different products and software solutions, for instance: VMware [37, 42], Xen [38] or VirtualBox [39]. Additionally, the processor manufacturers are also interested in virtualization and they are providing specific solutions for virtualization in their microchips (Intel VT-x[1] technology and AMD-V Pacifica [2]). Thanks to this new technology, the paravirtualization and native virtualization solutions can improve its performance and features.

[1] `http://www.intel.com/technology/virtualization/`

[2] `http://www.amd.com/us-en/0,,3715_15781,00.html?redir=wsv08`

To sum up, the virtualization is a promising technology that can improve the deployment and maintenance of clusters, due to its main features: resource isolation, guest OS instantiation and snapshots. On the other hand, take into account research groups or institutions that have clusters or additionally servers dedicated to other services like HTTP, FTP, etc. Despite of the load of those machines, it could be interesting to take advantage of the computing resources that these machines can add to our cluster by means of VMs. The goal is to install virtual cluster nodes on non-dedicated cluster nodes to obtain computing resources when requested and available.

To the best knowledge of the authors, several attempts have been done on integrating virtualization and clusters. J.S. Chase et. al. [43] present new mechanisms for dynamic resource management in a cluster manager called Cluster-On-Demand. I. Foster et. al. [44] propose to give custom client clusters to circumvent the hardware and software heterogeneity of clusters. Finally, W. Emeneker et. al. [45] propose to use virtualization in clusters for job forwarding and spanning. However, none of the above cited articles use Dynamite, the load balancing tool employed on this work. Therefore, what we are considering in the context of the problem presented in this chapter is the deployment of additional cluster nodes on-demand by means of virtualization. The goal is to improve Dynamite (see Section 5.4) adding a new virtualization feature that can request more computing power by launching virtual nodes on other machines, for example other servers from the institution. The application will determine when it is necessary more computer power and request it by launching those virtual cluster nodes.

This improvement to Dynamite will benefit the whole cluster as virtualization gives identical cluster computing nodes as real ones. It is obvious that using a virtualization technology gives some overhead [38, 37], but thanks to new microprocessors from Intel and AMD and the improvements on the technology, virtual nodes performance is more or less equal to real hardware [38, 37].

Additionally, the use of virtualization technology will allow to checkpoint and migrate any running virtual cluster node without having to implement any special library for checkpointing and migration at level process like Overeinder et. al. proposed on his work [30]. Virtualization simplifies this problem by checkpointing the whole node and migrating it without losing any kind of connectivity or data (see [38]). Furthermore, the possibility of serializing and migrating a VM (snapshots) opens new opportunities to load balancing and reliability.

Other benefit of using VMs is the possibility of running cluster nodes on different OSs and architectures because virtualization abstracts the underlying hardware. Thus, it will be possible to run GNU/Linux cluster nodes on Microsoft Windows machines without any kind of problem. Furthermore, thanks to virtualization it will be possible to harness all the computers from an institution or research laboratory independently of its OS platform, providing more computing power to the cluster when necessary. Bear in mind that Dynamite will launch virtual machines only when more computing power is required.

In conclusion, thanks to this new approach it will be possible to have a more powerful and flexible cluster that auto-adapts itself to the CPU load launching or stopping virtual nodes on-demand.

7 Conclusions

This chapter has presented the modeling of a well-known complex system, the laser, by means of a parallel version of a bioinspired algorithm, the cellular automaton.

By means of a series of experiments, we have considered key factors of the parallel algorithm when running on clusters of workstation. We have used a cluster computing environment for being better suited in general than a grid computing platform to run a parallel CA due to its lower latency on the communications.

Firstly we have shown the feasibility of CA for modeling the laser. Secondly, we have studied the performance obtained by a parallel version of the model, and finally we have considered the execution of the algorithm on a non-dedicated cluster, when external loads dynamically arrive while the CA tasks are being simultaneously run.

We have thus shown that the parallel version of the algorithm –following a master-worker model using the message passing mechanism- can offer good scalability when running on a cluster, which is of interest for running large versions of the CA model for modeling realistic laser systems.

We have then moved to a more realistic scenery by considering the presence of external loads on the cluster system. We have evaluated the performance of the application including artificial external loads to simulate the effect of other tasks running simultaneously on the cluster. In this case, a dynamic load balancing strategy has been used, with two main differences with respect to most previous parallel CA implementations: load can be migrated to new nodes initially not belonging to the pool and the load balancing functionality is uncoupled from the CA algorithm. For this purpose, we have run the parallel application on top of a dynamic load balancing software tool –Dynamite–. This modular approach has the advantage that changes can be introduced to the CA algorithm or to the dynamic load balancing strategy without perturbing each other.

In spite that for this kind of application –a synchronous cellular automaton– all the computing nodes must have finished an iteration before the next one can be initiated, the results have been very satisfactory. The performance of the parallel application is improved by the load balancing strategy from 60% to 80% when there are some idle nodes on the cluster to which some load can be migrated. Still when there were no such idle nodes, the execution time was always shorter than without the use of load balancing.

In conclusion, we have reviewed the application of a parallel cellular automata model to simulate laser dynamics and we have also presented evidence of the feasibility of running large parallel simulations using this approach to simulate realistic laser devices on heterogeneous non-dedicated clusters if an adequate dynamic load balancing strategy is used.

References

1. Einstein, A.: Zur quantenmechanik der strahlung. Physikalische Zeitschrift 18, 121–128 (1917)
2. Siegman, A.E.: Lasers. University Science Books (1986)
3. Guisado, J.L., Jiménez-Morales, F., Guerra, J.M.: Cellular automaton model for the simulation of laser dynamics. Physical Review E 67(6), 66708 (2003)
4. Ilachinski, A.: Cellular automata. A discrete Universe. World Scientific, Singapore (2001)
5. Sloot, P.M.A., Hoekstra, A.G.: Modeling Dynamic Systems with Cellular Automata, ch. 21, pp. 21–1+6. Chapman & Hall/CRC, Boca Raton (2007)
6. Chopard, B., Droz, M.: Cellular Automata Modeling of Physical Systems. Cambridge University Press, Cambridge (1998)
7. Guisado, J.L., Jiménez-Morales, F., Guerra, J.M.: Application of shannon's entropy to classify emergent behaviors in a simulation of laser dynamics. Mathematical and Computer Modelling 42, 847–854 (2005)
8. Guisado, J.L., Jiménez-Morales, F., Guerra, J.M.: Computational simulation of laser dynamics as a cooperative phenomenon. Physica Scripta 118, 148–152 (2005)
9. Guisado, J.L., Jiménez-Morales, F., Fernández de Vega, F.: Cellular automata and cluster computing: An application to the simulation of laser dynamics. Advances in Complex Systems 10(Suppl.1), 167–190 (2007)
10. Guisado, J.L., Fernández de Vega, F., Jiménez-Morales, F., Iskra, K.: Parallel implementation of a cellular automaton model for the simulation of laser dynamics. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3993, pp. 281–288. Springer, Heidelberg (2006)
11. Guisado, J.L., Fernández de Vega, F., Iskra, K.: Performance analysis of a parallel discrete model for the simulation of laser dynamics. In: 2006 International Conference on Parallel Processing, Workshops, pp. 93–99. IEEE Computer Society, Los Alamitos (2006)
12. Guisado, J.L., Fernández de Vega, F., Jiménez-Morales, F., Iskra, K.A., Sloot, P.M.A.: Using cellular automata for parallel simulation of laser dynamics with dynamic load balancing. International Journal of High Performance Systems Architecture 1(4), 251–259 (2009)
13. Talia, D.: Cellular processing tools for high-performance simulation. IEEE Computer 33(9), 44–52 (2000)
14. Resnick, M.: Turtles, Termites, and Traffic Jams. MIT Press, Cambridge (1994)
15. Cannataro, M., Di Gregorio, S., Rongo, R., Spataro, W., Spezzano, G., Talia, D.: A parallel cellular automata environment on multicomputers for computational science. Parallel Computing 21(5), 803–823 (1995)
16. Spezzano, G., Talia, D., Di Gregorio, S., Rongo, R., Spataro, W.: A parallel cellular tool for interactive modeling and simulation. IEEE Computational Science & Engineering 3(3), 33–43 (1996)
17. Hutchinson, D., Kattner, L., Lanthier, M., Maheshwari, A., Nussbaum, D., Roytenberg, D., Sack, J.R.: Parallel neighbourhood modeling: research summary. In: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, pp. 204–207 (1996)
18. Carotenuto, L., Mele, F., Furnari, M., Napolitano, R.: PECANS: A parallel environment for cellular automata modeling. Complex Systems 10(1), 23–42 (1996)

19. Zeigler, B., Moon, Y., Kim, D., Ball, G.: The DEVS environment for high-performance modeling and simulation. IEEE Computational Science & Engineering 4(3), 61–71 (1997)
20. Schoneveld, A., de Ronde, J.F.: P-CAM: a framework for parallel complex systems simulations. Future Generation Computer Systems 16(2), 217–234 (1999)
21. Toffoli, T., Margolus, N.: Cellular automata machines: a new environment for modeling. MIT Press, Cambridge (1987)
22. Sloot, P.M.A., Kaandorp, J.A., Hoekstra, A.G., Overeinder, B.J.: Distributed simulation with cellular automata: architecture and applications. In: Bartosek, M., Tel, G., Pavelka, J. (eds.) SOFSEM 1999. LNCS, vol. 1725, pp. 203–248. Springer, Heidelberg (1999)
23. D'Ambrosio, D., Spataro, W.: Parallel evolutionary modeling of geological processes. Parallel Computing 33(3), 186–212 (2007)
24. Mazzariol, M., Gennart, B., Hersch, R.: Dynamic load balancing of parallel cellular automata. In: Proc. SPIE Conference on Parallel and Distributed Methods for Image Processing IV, San Diego, July 2000, vol. 4118, p. 2129. SPIE (2000)
25. Kohring, G.A.: Dynamic load balancing for parallelized particle simulations on MIMD computers. Parallel Computing 21, 683–693 (1995)
26. Cortés, A., Planas, M., Millán, J.L., Ripoll, A., Senar, M.A., Luque, E.: Applying load balancing in data parallel applications using DASUD. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 237–241. Springer, Heidelberg (2003)
27. Fabero, J.C., Martin, I., Bautista, A., Molina, S.: Dynamic load balancing in a heterogeneous environment under PVM. In: 4th Euromicro Workshop on Parallel and Distributed Processing (PDP 1996), pp. 414–419. IEEE Computer Society, Los Alamitos (1996)
28. Weimar, J.R.: Cellular automata for reaction-diffusion systems. Parallel Computing 23(11), 1699–1715 (1997)
29. Dick van Albada, G., Clinckmaillie, J., Emmen, A.H.L., Gehring, J., Heinz, O., van der Linden, F., Overeinder, B.J., Reinefeld, A., Sloot, P.M.A.: Dynamite - blasting obstacles to parallel cluster computing. In: Sloot, P.M.A., Hoekstra, A.G., Bubak, M., Hertzberger, B. (eds.) HPCN-Europe 1999. LNCS, vol. 1593, pp. 300–310. Springer, Heidelberg (1999)
30. Overeinder, B.J., Sloot, P.M.A., Heederik, R.N., Hertzberger, L.O.: A dynamic load balancing system for parallel cluster computing. Future Generation Computer Systems 12(1), 101–115 (1996)
31. Iskra, K., Hendrikse, Z.W., Dick van Albada, G., Overeinder, B.J., Sloot, P.M.A., Gehring, J.: Experiments with migration of message-passing tasks. In: Buyya, R., Baker, M. (eds.) GRID 2000. LNCS, vol. 1971, pp. 203–213. Springer, Heidelberg (2000)
32. Iskra, K., Hendrikse, Z.W., Dick van Albada, G., Overeinder, B.J., Sloot, P.M.A.: Dynamic migration of PVM tasks. In: ASCI 2000, Proceedings of the sixth annual conference of the Advanced School for Computing and Imaging, June 2000, pp. 206–212 (2000)
33. Folino, G., Spezzano, G.: An autonomic tool for building self-organizing grid-enabled applications. Future Generation Computer Systems 23(5), 671–679 (2007)
34. Vadhiyar, S.S., Dongarra, J.J.: Self adaptivity in grid computing. Concurrency Computation Practice and Experience 17(2-4), 235–257 (2005)
35. Foster, I.: Designing and building parallel programs. Addison-Wesley, Reading (1995)
36. Dongarra, J., Foster, I., Fox, G.C., Gropp, W., Kennedy, K., Torczon, L., White, A. (eds.): Sourcebook of parallel computing. Morgan Kaufmann, San Francisco (2003)
37. Sugerman, J., Venkitachalam, G., Lim, B.: Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor

38. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 164–177 (2003)
39. Watson, J.: Virtualbox: bits and bytes masquerading as machines. Linux J. 2008(166), 1 (2008)
40. Elnozahy, E., Alvisi, L., Wang, Y., Johnson, D.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys (CSUR) 34(3), 375–408 (2002)
41. Robin, J., Irvine, C.: N.P.S.M.C.D.O.C. SCIENCE. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, Defense Technical Information Center (2000)
42. Nieh, J., Leonard, O.C.: Examining VMware. j-DDJ 25(8), 70, 72–74, 76 (2000)
43. Chase, J.S., Irwin, D.E., Grit, L.E., Moore, J.D., Sprenkle, S.E.: Dynamic Virtual Clusters in a Grid Site Manager. In: HPDC 2003: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, p. 90 (2003)
44. Foster, I., Freeman, T., Keahy, K., Scheftner, D., Sotomayer, B., Zhang, X.: Virtual Clusters for Grid Communities. In: CCGRID 2006: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, pp. 513–520 (2006)
45. Emeneker, W., Stanzione, D.: Dynamic Virtual Clustering, 2007. In: IEEE International Conference on Cluster Computing, pp. 84–90 (2007)

Author Index