Improving Extremal Optimization in Load Balancing by Local Search

Ivanoe De Falco[1], Eryk Laskowski[2](✉), Richard Olejnik[3], Umberto Scafuri[1], Ernesto Tarantino[1], and Marek Tudruj[2,4]

[1] Institute of High Performance Computing and Networking, CNR, Naples, Italy
{ivanoe.defalco,umberto.scafuri,ernesto.tarantino}@na.icar.cnr.it
[2] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
{laskowsk,tudruj}@ipipan.waw.pl
[3] Computer Science Laboratory, University of Science and Technology of Lille, Villeneuve-d'Ascq, France
richard.olejnik@lifl.fr
[4] Polish-Japanese Institute of Information Technology, Warsaw, Poland

Abstract. The paper concerns the use of Extremal Optimization (EO) technique in dynamic load balancing for optimized execution of distributed programs. EO approach is used to periodically detect the best candidates for task migration leading to balanced execution. To improve the quality of load balancing and decrease time complexity of the algorithms, we have improved EO by a local search of the best computing node to receive migrating tasks. The improved guided EO algorithm assumes a two-step stochastic selection based on two separate fitness functions. The functions are based on specific program models which estimate relations between the programs and the executive hardware. The proposed load balancing algorithm is compared against a standard EO-based algorithm with random placement of migrated tasks and a classic genetic algorithm. The algorithm is assessed by experiments with simulated load balancing of distributed program graphs and analysis of the outcome of the discussed approaches.

Keywords: Distributed program design · Extremal optimization · Load balancing

1 Introduction

The paper presents Extremal Optimization (EO) [1] based load balancing algorithm for distributed systems. The proposed algorithm is composed of iterative optimization phases which improve program task placement on processors to determine the possibly best balance of computational loads and to define periodic migration of tasks. The EO algorithm discovers the candidate tasks for migration based on a special quality model including the computation and communication parameters of parallel tasks. The paper presents an improved load balancing algorithm comparing the algorithm given in [2], which was based on classical Extremal Optimization approach. In the classical EO the fully random selection of a new improved partial solution in the neighbourhood of the solution being modified is done. The fully random selection has been considered unsatisfactory, since for a big number of executive processors a degradation of the quality of obtained result (the parallel speedup of the applications) was observed. Therefore, we have improved the applied EO algorithm by a replacement of the fully random selection of the target computing node in migration by the stochastic selection performed with the guidance by some knowledge of the problem properties. The guidance is based on a formula which estimates how a migrated task matches the given processor in respect to the global computational and communicational balance in the system. It should be stressed that we have maintained the nature-inspired solution improvement but done in the way which speeds up the convergence of the algorithm. As a result we have obtained a correct behavior of the algorithm when the cardinality of processor set in the system increases.

The algorithm is assessed by experiments with simulated load balancing of distributed program graphs. In particular, the experiments compare three algorithms: the proposed load balancing method including the EO with a guided stochastic selection of the improved solution, an EO with fully random selection of the improved solution and a genetic algorithm (GA). The comparison shows that the quality of load balancing with the guided EO is in most cases better than with fully random selection and with the GA.

The paper is organized as follows. In Section 2 the related works in load balancing based on nature inspired algorithms are reported. In Section 3 the EO principles are shortly explained, and the EO with guided state changes is introduced. Section 4 describes the theoretical foundations for the discussed algorithm, explains how the EO is applied to the dynamic processor load balancing. In Section 5 the experiments which assess the proposed algorithms are presented.

2 Related Works

A huge quantity of papers exist in literature dealing with dynamic load balancing in parallel and distributed systems. Good reviews and classifications of classic load balancing methods are presented in [3–6].

Genetic algorithms have been the first nature–inspired optimization method to be used with reference to this issue. Munetomo et al. [7] are among the first to present a genetic algorithm for stochastic environments and show its application to dynamic load balancing in distributed systems. Zomaya and Teh [8] investigate how a genetic algorithm can be employed to solve the dynamic load balancing problem. To address the problem of dynamic load balancing in a processing pool, Uyar and Harmanci [9] apply an improved genetic algorithm called *damGA* (diploidy-aging-meiosis Genetic Algorithm). Very recently, Lin and Deung [10] face dynamic load balancing in cloud-based multimedia system using a genetic algorithm. More recently, other nature–inspired optimization methods have been investigated for dynamic load balancing, including Particle

Swarm Optimization (PSO). A good review of several such methods can be found in a very recent paper [11].

At the best of our knowledge, no other authors have attempted to use EO for dynamic load balancing. We feel, instead, that EO has all the desired features useful to efficiently tackling this problem. Firstly, EO is perfectly suited to face combinatorial optimization problems where solutions are represented by integer values. Secondly, evaluating each component of a solution on its own and changing a bad component only, rather than the whole solution, is highly desirable when an incremental improvement is necessary. GA or PSO would modify the solution as a whole, possibly destroying good issues too. So, the proposed approach has clear originality features and enables making profit of EO advantages such as low computational complexity and limited use of memory space.

3 Extremal Optimization Algorithm Principles

Extremal Optimization was proposed by Boettcher and Percus [1], following the Bak–Sneppen approach of self–organized dynamic criticality [12]. It represents a method for NP–hard combinatorial and physical optimization problems. EO is based on improvements of a single solution S consisting of a given number of components s_i, called species. Each component is a variable of the problem. A local fitness value is assigned to each component. At each time step, S is evolved by randomly updating the worst variable only in respect to ϕ_i, to a solution S' belonging to its neighbourhood $Neigh(S)$. After each update, a global fitness $\Phi(S)$ is computed and the modified solution S' is registered if its global fitness is better than that of the best solution found so far.

We apply a probabilistic version of EO based on a parameter τ, i.e., τ–EO, introduced by Boettcher and Percus, which prevents the solutions from staying in a local optimum. For a minimization problem, the components are first ranked in the increasing order of local fitness values. Then, a distribution probability k over ranks is considered as follows: $p_k \sim k^{-\tau}$, $1 \leq k \leq |S|$ for a given value of τ. At each update of S, a rank k is selected according to p_k so that the species s_i with $i = \pi(k)$ randomly changes its state and the solution moves unconditionally to $S' \in Neigh(S)$.

3.1 Extremal Optimization With Guided State Changes

During our experimental research on load balancing of distributed applications, reported in [2], we have revealed that EO is able to provide the best results for almost all combinations of system and application parameters.

However, we have noticed that, when the number of neighbour states of rank k increases (i.e. the number of processors is higher), the algorithm starts struggling with the problem of too many possible moves. The probability of "good" state change decreases. To alleviate this problem we incorporate more problem-specific information into the algorithm. It is implemented as a local target function ω_s, which is computed for all neighbours $Neigh(S)$ of rank k. Then

Algorithm 1. τ–EO algorithm with Guided State Changes (EO–GS)

```
initialize configuration $S$ at will
$S_{\text{best}} \leftarrow S$
while total number of iterations $\mathcal{N}_{\text{iter}}$ not reached do
    evaluate $\phi_i$ for each variable $s_i$ of the current solution $S$
    rank the variables $s_i$ based on their fitness $\phi_i$
    choose the rank $k$ according to $k^{-\tau}$ so that the variable $s_j$ with $j = \pi(k)$ is selected
    evaluate $\omega_s$ for each neighbour $s' \in Neigh(S)$, generated by $s_j$ change
    rank neighbours $s' \in Neigh(S)$ based on the value of target function $\omega_s$
    choose $S' \in Neigh(S)$ according to the exponential distribution $\text{Exp}(\lambda)$
    accept $S \leftarrow S'$ unconditionally
    if $\Phi(S) < \Phi(S_{\text{best}})$ then
        $S_{\text{best}} \leftarrow S$
    end if
end while
return $S_{\text{best}}$ and $\Phi(S_{\text{best}})$
```

the neighbours are sorted according to the increasing value of ω_s. The new state $S' \in Neigh(S)$ is selected randomly using the exponential distribution $\text{Exp}(\lambda)$ over the sorted neighbours $Neigh(S)$. Thus, the stochastic local search towards "better" neighbours (according to the value of ω_s) is performed. The bias to the "better" values is controlled by the λ parameter of the exponential distribution. The scheme of the Extremal Optimization with Guided State Changes (EO–GS) is shown in Algorithm 1.

4 Load Balancing Based on Extremal Optimization

The proposed load balancing algorithm is meant for distributed application programs composed of T indivisible tasks which are threads (single-thread processes). Each task is composed of sequences of computational instructions (blocks) separated by communication instructions with other tasks.

We assume a centralized program execution environment which means that the executive system works under control of some load balancing infrastructure responsible for organizing optimized execution of programs. The executive system is a cluster of N processor aka computational nodes interconnected by a message passing network.

Our load balancing problem is formally defined in the following way: during program execution dynamically map each task t_k, $k \in \{1 \ldots |T|\}$ of the program to a computational node n, $n \in [0, N-1]$ in such a way that the total program execution time is minimized, assuming the program and system definition as stated earlier in this section. Dynamic task mapping to computational nodes can change during program execution by means of task migration.

The load balancing method proposed in the paper, consists in execution of a series of indivisible pairs of two main steps: the detection and the correction of processor load imbalance. The load imbalance detection step employs some measurement infrastructure to monitor the states of the executive system and the application program relevant for the detection of system load imbalance. In parallel with the execution of an application program, computing nodes periodically report their loads to a load balancing monitor which evaluates the current system load imbalance value. Depending on this value, the second step (i.e. the imbalance correction) is done or step one is repeated. In the second step, we execute the EO-based algorithm described in next sections, which determines the set of tasks for migration and the migration target nodes. Based on that, the physical task migrations are executed and the algorithm goes to step one.

4.1 Detection of Load Imbalance

Two parameters are used to evaluate the state of the system:
$Ind_{power}(n)$ – computing power of a processor node n, which is the sum of nominal computing powers of all cores on the node, in MIPS, MFLOPS or similar,
$Time^{\%}_{\mathrm{CPU}}(n)$ – the current CPU time availability i.e. percentage of the CPU computing power currently available for application threads on the node n, periodically estimated by load observation agents on computing nodes.

A load imbalance LI (a boolean) is defined based on the difference of the current CPU time availability between the most heavily and the least heavily loaded computing nodes:

$$LI = \max_{n \in P}(Time^{\%}_{\mathrm{CPU}}(n)) - \min_{n \in P}(Time^{\%}_{\mathrm{CPU}}(n)) \geq \alpha$$

where P is the set of all computing nodes. The detection of load imbalance equal *true* requires a load correction. α is determined using an experimental approach (in our experiments we have set it between 25% and 75%).

4.2 Correction of Load Imbalance

The application is characterized by two metrics, which should be provided by a programmer based on the volume of computations and communications in tasks:

1. $\mathrm{COM}(t_s, t_d)$ is the communication metrics for a pair of tasks t_s and t_d,
2. $\mathrm{WP}(t)$ is the load weight metrics introduced by a task t.

$\mathrm{COM}(t_s, t_d)$ and $\mathrm{WP}(t)$ metrics can constitute exact values, e.g. for well-defined tasks sizes and inter-task communication in regular parallel applications, or only some predictions, e.g. when the computation depends on the processed data as in irregular parallel applications.

A task mapping solution S is represented by a vector $\mu = (\mu_1, \ldots, \mu_{|T|})$ of $|T|$ integers from the interval $[0, N-1]$, where the value $\mu_i = j$ means that the solution S under consideration maps the i–th task t_i of the application onto the computing node j.

The global fitness function $\Phi(S)$ is defined as

$$\begin{aligned}\Phi(S) = {} & attrExtTotal(S) * \Delta_1 + migration(S) * \Delta_2 + \\ & + imbalance(S) * [1 - (\Delta_1 + \Delta_2)]\end{aligned} \quad (1)$$

where Δ_1, Δ_2 parameters control the weight of components of the global fitness, $1 > \Delta_1 \geq 0$, $1 > \Delta_2 \geq 0$ and $\Delta_1 + \Delta_2 < 1$. The function $\mathit{attrExtTotal}(S) \in \{0, 1\}$ represents the impact of the total external communication between tasks on the quality of a given mapping S. The function $\mathit{migration}(S) \in \{0, 1\}$ is a migration costs metrics. It is equal to 0 when there is no migration, when all tasks have to be migrated $\mathit{migration}(S) = 1$. The function $\mathit{imbalance}(S) \in \{0, 1\}$ represents the numerical load imbalance metrics in the solution S. It is equal to 1 when there exists at least one unloaded computing node, otherwise it is equal to the normalized average absolute load deviation of tasks in S.

The local fitness function of a task $\phi(t)$ is designed in such a way that it forces moving tasks away from overloaded nodes, at the same time preserving low external (inter-node) communication. The γ parameter $(0 < \gamma < 1)$ allows tuning the weight of load metrics.

$$\phi(t) = \gamma * \mathit{load}(\mu_t) + (1 - \gamma) * \mathit{rank}(t) \tag{2}$$

The function $\mathit{load}(n)$ indicates whether the node n, which executes t, is overloaded (i.e. it indicates how much its load exceeds the average load of all nodes). The $\mathit{rank}(t)$ function governs the selection of best candidates for migration. The chance for migration have tasks, which show low communication with their current node (attraction) and low load deviation from the average load. The load balancing parameters mentioned above are explained in full details in [2].

4.3 Guided Target Node Selection for State Changes

In the standard EO algorithm (see [2]), any neighboring state could be selected randomly using the uniform probability distribution. The idea of a guided state changes is based on some "biased" random selection, to enable preferring some neighbors over others. At each update of rank k, nodes $n \in N$ are sorted according to $\omega(n1, n2)$ function and one of them is selected using the exponential distribution $\mathrm{Exp}(\lambda)$. The bias to the "better" values, i.e. lower values of $\omega(n1, n2)$ in our case, is controlled by the λ parameter of the exponential distribution.

A "biased" random selection uses formula similar to those used for the local fitness calculation to qualify the computing nodes for migration of task j:

$$\omega(n1, n2) = \begin{cases} \mathit{relload}(n1) - \mathit{relload}(n2) & \text{if } \mathit{relload}(n1) \neq \mathit{relload}(n2) \\ \mathit{attrext}(j, n2) - \mathit{attrext}(j, n1) & \text{otherwise} \end{cases}$$

where:

$$\mathit{attrext}(j, n) = \sum_{e \in T(n)} (\mathrm{COM}(e, j) + \mathrm{COM}(j, e)), \ \text{ normalized vs. } \max_{e \in N}(\mathit{attrext}(j, e))$$

$$\mathit{relload}(n) = \frac{\mathit{loaddev}(n) - \min_{m \in [0, N-1]} \mathit{loaddev}(m)}{\max_{m \in [0, N-1]} \mathit{loaddev}(m) - \min_{m \in [0, N-1]} \mathit{loaddev}(m)}$$

$$\mathit{loaddev}(n) = \mathrm{NWP}(S, n)/\mathit{Ind}_{power}(n) - \overline{\mathrm{WP}}$$

and $T(n) = \{t \in T : \mu_t = n\}$ — the set of threads, placed on node n, $\mathrm{NWP}(S, n) = \sum_{t \in T : \mu_t = n} \mathrm{WP}(t)$, $\overline{\mathrm{WP}} = \sum_{t \in T} \mathrm{WP}(t) / \sum_{n \in [0, N-1]} Ind_{power}(n)$.

When $\omega(n1, n2)$ has a low value, the computational load of node $n1$ is lower than that of node $n2$ or the task j has stronger attraction to node $n1$. This is the preferred target of migration. High values of $\omega(n1, n2)$ indicate overloading of node $n1$ or no communication to this node from task j.

5 Experimental Results

We describe below experimental results obtained by simulated execution of application programs with the proposed method of load balancing in a distributed system. The assumed program parallelization model corresponds to parallelization based on message-passing, using the MPI library for communication. The experiments were run in a simulated cluster of multi–core processors. Each processor had its own main memory and a network interface. At the level of the network interfaces data transfers and communication contention were modeled.

In the experiments, a set of 10 randomly generated synthetic exemplary programs was used. Their general structures were phase-like, in which they resembled MPI-based parallel programs which corresponded to numerical computations or simulations of physical phenomena. The programs were represented as a set of phases (see Fig 1), each composed of parallel tasks (threads). Tasks of the same phase could communicate. At the boundaries between phases there was a global exchange of data which corresponded to external communication between processes. Application programs contained from about 60 to 550 tasks. Their communication/computation ratio C/E was in the range $[0.05, 0.20]$.

Based on the time properties of tasks two types of applications were distinguished: regular and irregular. Regular applications had fixed task execution times. Irregular applications had the execution time of tasks depending on the processed data. They showed unpredictable both execution times of tasks and the communication schemes. With irregular tasks, system load imbalance could

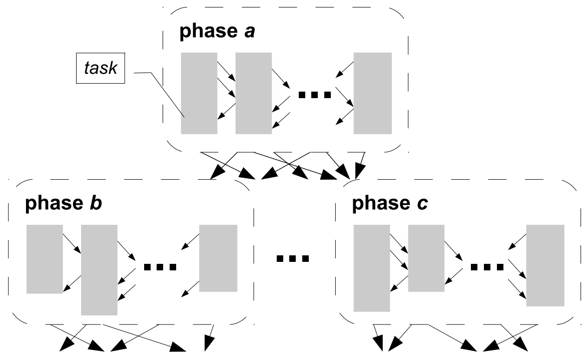

Fig. 1. The general structure of exemplary applications

occur even without variations in computing nodes availability. With regular applications system load imbalance could occur due to the suboptimal placement of tasks on processors or when runtime conditions had changed. The properties of the proposed load balancing algorithm for both types of applications were comparatively examined.

For comparison purposes, the same simulated parallel environment and the set of graphs were used. We compared EO and EO–GS to genetic algorithm (GA) which used the same global fitness function. GA used binary-encoded chromosomes, in which an allele at position i was the processor number of the task i. Two genetic operators were used: single-point crossover and mutation. The selection was based on roulette-wheel scheme. We used the following GA parameters: the size of population – 50, the probability of mutation – 0.015, the probability of crossover – 0.25, the number of iterations – 500. Half of the chromosomes of the initial population was generated randomly, the second half was initialized through cloning of the current placement of application tasks.

5.1 Performance of the Presented Algorithms

In the first series of experiments, load-balanced execution of phase-like applications was studied in systems containing from 2 to 32 homogeneous processor nodes. The following parameters for load balancing control were used: $\alpha = 0.5, \Delta_1 = 0.25, \Delta_2 = 0.25, \gamma = 0.5, \tau = 1.5$, for EO–GS $\lambda = 1.0$. The number of iterations for EO and EO–GS was set to 500. The results correspond to averages of 5 runs of each application. For each run 4 different methods of initial task placements (random, round-robin, METIS, packed) were tested. METIS is a graph partitioning optimization software [13]. The packed method consists in round-robin mapping of equal groups of tasks. In total, 20 runs were executed for each parameter set to produce an averaged result.

The speedup of both EO–based algorithms and the genetic algorithm as a function of the number of processors is shown in Fig. 2. For regular applications (upper curves) the speedup improvement due to EO–based algorithms is generally bigger (not worse or better) than that of GA. Our exemplary irregular applications (lower curves) give smaller speedup than regular ones (with or without load balancing) what is an expected result, since parallel execution of such applications is less efficient. However, for irregular applications the EO-GS algorithm is generally the best comparing all the others. It should be stressed that EO-GS gives much better results than EO and GA especially for a bigger number of processors. It is due to completely random placement of migrated tasks on processors in EO and GA, not supported by any knowledge of the system and program state. EO-GS uses a more thorough migration target selection.

Since migration costs can be very different (a single migration can be as short as a simple task activation message, but also it can involve a transfer of the processed data, which is usually very costly), we decided to keep the generality of our experiment results and to approximate the imposed load balancing costs by the number of task migrations, Fig. 3. The number of migrations is decidedly higher for irregular applications (upper curves). The average cost imposed by EO–GS algorithm is generally lower than the cost introduced by other approaches. For irregular applications the migration number with EO-GS is lower than with the EO and GA. For regular applications the number of task migrations in both EO-based algorithms is almost halved comparing GA. Experiments revealed that the GA approach can not work out an efficient migration decision for irregular applications run on bigger number of processor, thus we notice sudden drop in the GA (irg) curves both in Fig. 2 and 3.

To generalize comparisons of performance of the discussed load balancing algorithms, we have computed the average speedup improvement of the considered

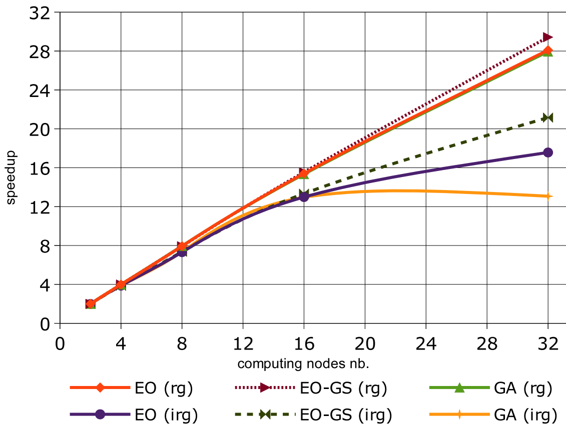

Fig. 2. Speedup for different number of nodes for tested algorithms

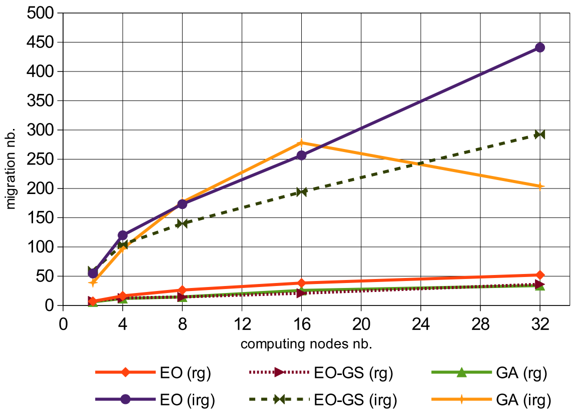

Fig. 3. Cost of the dynamic load balancing as the number of task migrations per single execution of an application

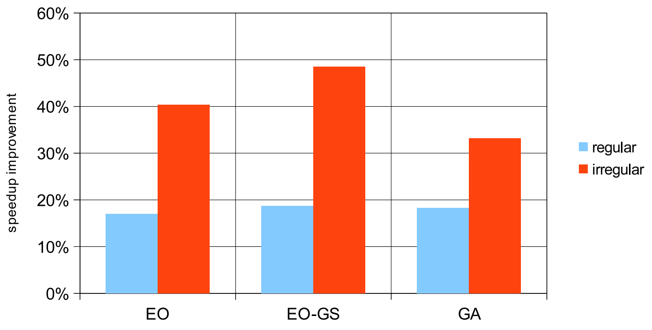

Fig. 4. Average speedup improvement for different algorithms due to load balancing

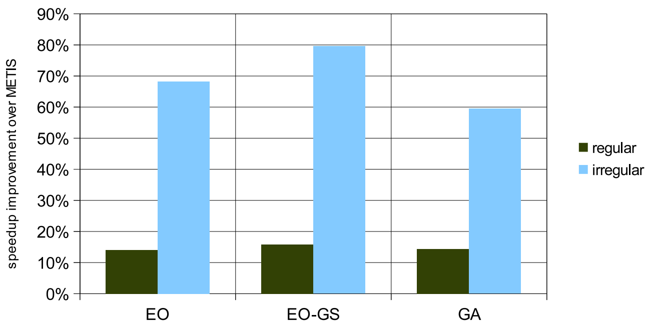

Fig. 5. Comparison of speedup obtained by different algorithms and METIS initial task placement

algorithms over execution without load balancing. The speedup improvement is calculated as $S_b/S_u - 1$, where S_b is the speedup obtained when load balancing algorithm is active, S_u is the speedup of the unbalanced execution. The best speedup improvement over the unbalanced execution for both irregular and regular applications is provided by EO–GS algorithm (see Fig. 4).

To justify the quality of the results, we have compared the speedup obtained for dynamic load balancing using the analysed algorithms to the speedup based on static task placement obtained by METIS graph partitioning algorithm. To do so, we executed regular and irregular applications with initial task placement by METIS and the same applications starting from imbalanced, random initial placement with the dynamic load balancing switched on. For regular applications the improvement due to load balancing with static initial METIS placement is small (in the range 12% – 16%, see Fig. 5). The improvement indicates that the compared algorithms are able to work out profitable migration decisions even

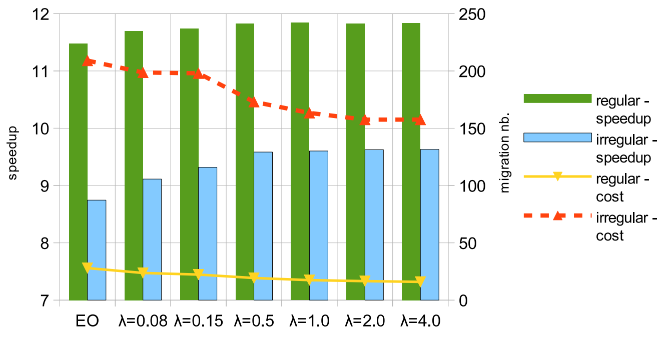

Fig. 6. Average application speedup and migration number (load balancing cost) for different values of EO–GS parameters as a function of λ

after METIS initial optimisation of regular applications, resulting in their balanced execution. For irregular applications METIS initial optimisation is not sufficient for efficient balanced execution up to the end of their task sets. For irregular applications speedup improvement after METIS initialisation due to dynamic load balancing is on average several times higher than for regular applications. We can see that the EO–GS algorithm gives here the best results, better than other studied algorithms by 15%.

5.2 The Algorithm Parameter Setting

The influence of the setting of λ parameter on overall performance of EO–GS algorithm is shown in Fig. 6 (EO denotes here the results for the standard EO algorithm). Increasing value of λ results in a noticeable increase of the speedup for irregular applications, at the same time reducing the cost of load balancing (i.e. the number of migrations). Although the cost initially decreases slowly, for $\lambda = 0.5$ or more is much smaller than in the standard EO algorithm. For regular applications λ has almost no impact on the average speedup (there is a slight increase) and slightly reduces the number of migrations. Note that regular applications show already high speedup for standard EO, thus improvement is possible only through reduction of the number of migrations. For both types of graphs increasing λ above 1.0 has no longer a significant effect on the results.

6 Conclusions

The paper has presented the dynamic load balancing in distributed systems based on application of the Extremal Optimization approach. The proposed load balancing algorithm is an improved version of the classic Extremal Optimization, in which we replaced the completely random computing node selection by the stochastic selection where node selection probability is guided by some knowledge of the problem. Our approach proved to be an efficient method for load balancing, distinguished by low computational complexity and limited use of memory space.

The proposed algorithm has been assessed by experiments with simulated load balancing of distributed program graphs. In particular, the experiments compare load balancing with EO with guided search against the classic EO and genetic algorithm based on equivalent theoretical foundations. The comparison shows that the quality of the improved EO-based load balancing outperforms in most cases that with classical EO and the genetic algorithm.

References

1. Boettcher, S., Percus, A.G.: Extremal optimization: methods derived from coevolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999), pp. 825–832. Morgan Kaufmann, San Francisco (1999)
2. Olejnik, R., De Falco, I., Laskowski, E., Scafuri, U., Tarantino, E., Tudruj, M.: Load Balancing in Distributed Applications Based on Extremal Optimization. In: Esparcia-Alcázar, A.I. (ed.) EvoApplications 2013. LNCS, vol. 7835, pp. 52–61. Springer, Heidelberg (2013)
3. Barker, K., Chrisochoides, N.: An evaluation of a framework for the dynamic load balancing of highly adaptive and irregular parallel applications In: Proceedings of the ACM/IEEE Conference on Supercomputing, Phoenix. ACM Press (2003)
4. Willebeek-LeMair, M.H., Reeves, A.P.: Strategies for dynamic load balancing on highly parallel computers. IEEE Trans. on Parallel and Distributed Systems 4, 979–993 (1993)
5. Xu, C., Francis, C., Lau, M.: Load balancing in parallel computers: Theory and Practice. Kluwer Academic Publishers, Norwell (1997)
6. Khan, R.Z., Ali, J.: Classification of task partitioning and load balancing strategies in distributed parallel computing systems. International Journal of Computer Applications **60**(17), 48–53 (2012)
7. Munetomo, M., Takai, M.N.K., Sato, Y.: A stochastic genetic algorithm for dynamic load balancing in distributed systems. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, vol. 4, pp. 3795–3799. IEEE Press (1995)
8. Zomaya, A.Y., Teh, Y.-H.: Observations on using genetic algorithms for dynamic load-balancing. IEEE Trans. on Parallel and Distributed Systems 12(9), 899–911 (2001)
9. Uyar, A.S., Harmanci, A.E.: Application of an improved diploid genetic algorithm for optimizing performance through dynamic load balancing. In: Proceedings of 2002 WSEAS International Conferences. WSEAS Press (2002)
10. Lin, C.-C., Deng, D.-J.: Dynamic load balancing in cloud-based multimedia system using genetic algorithm. Chang, R.-S., et al (eds.) Advances in Intelligent Systems & Applications, SIST 20, pp. 461–470. Springer, Heidelberg (2013)
11. Mishra, M., Agarwal, S., Mishra, P., Singh, S.: Comparative analysis of various evolutionary techniques of load balancing: a review. International Journal of Computer Applications 63(15) (2013)
12. Sneppen, K., et al.: Evolution as a self-organized critical phenomenon. Proc. Natl. Acad. Sci. **92**, 5209–5213 (1995)
13. Karypis, G., Kumar, V.: Multilevel graph partitioning schemes. In: Proc. 24th Intern. Conf. Par. Proc., III. pp. 113–122. CRC Press (1995)