Automatic Parallelization of EC on GPGPUs and Clusters of GPGPU Machines with EASEA and EASEA-CLOUD

Pierre Collet, Frédéric Krüger, and Ogier Maitre

Abstract GPGPU cards are very difficult to program efficiently. This chapter explains how the EASEA and EASEA-CLOUD platforms can implement different evolution engines efficiently in a massively parallel way that can also serve as a starting point for more complex projects.

1 Introduction

Many papers show how GPGPU cards can be used to implement very fast massively parallel evolutionary algorithms. However, because these cards are quite complex to program, it is often very difficult to reproduce published results, because it is nearly impossible for authors to describe exactly how they implemented their algorithm. There are simply too many parameters and subtleties to consider.

Because hardware is involved in the performance of algorithms implemented on GPGPUs, details such as how individuals are implemented in memory may result in very different performance. For instance, coalescent memory calls are much faster than random memory accesses when several cores of a multiprocessor make memory calls. Supposing one tries to solve a problem coded with four floating-point values a, b, c, d. Storing the population as an array of individuals will result (in C) in the following memory layout:

$$\underbrace{a_0, b_0, c_0, d_0,}_{i_0}\underbrace{a_1, b_1, c_1, d_1,}_{i_1}\underbrace{a_2, b_2, c_2, d_2,}_{i_2}\underbrace{a_3, b_3, c_3, d_3,}_{i_3} \quad ...$$

P. Collet (✉) · F. Krüger · O. Maitre
ICUBE, University of Strasbourg, Illkirch, France
e-mail: Pierre.Collet@unistra.fr; Frederic.Kruger@etu.unistra.fr; Ogier.Maitre@unistra.fr

S. Tsutsui and P. Collet (eds.), *Massively Parallel Evolutionary Computation on GPGPUs*, Natural Computing Series, DOI 10.1007/978-3-642-37959-8_3,
© Springer-Verlag Berlin Heidelberg 2013

Now, supposing four SIMD cores running the same code, try to access the first parameter a of each individual they need to evaluate. The data that will be simultaneously read by the four cores will be the values of the variables a_0, a_1, a_2, a_3 which are not contiguous in memory.

However, if one had interleaved the different parameters of the population (as would have been the case if the array of individuals had been stored *à la* FORTRAN, for instance), the memory layout would have been the following:

$$a_0, a_1, a_2, a_3, \ldots b_0, b_1, b_2, b_3, \ldots c_0, c_1, c_2, c_3, \ldots d_0, d_1, d_2, d_3, \ldots$$

With this memory layout, the same four cores running the exact same code as before will end up with a much smaller execution time, because a much faster coalescent memory call will be performed.

Because of the difficulty to implement efficient GPGPU code and the difficulty to reproduce obtained results, in Strasbourg we have designed a massively parallel evolutionary platform that can automatically parallelize evolutionary algorithms in a standard predefined way. Whenever our team designs and publishes a new GPGPU algorithm, we take the effort to include the algorithm into the platform, therefore allowing anyone to benefit from this generic implementation to either:

- Implement a massively parallel evolutionary algorithm to solve a problem, using one of the available paradigms, or
- Implement a basic massively parallel evolutionary algorithm that is implementing one of the available paradigms, and use the produced source code as a starting point for a specific development.

This chapter rapidly describes the EASEA language and platform [18] for the reader who would like to benefit from the computing power of GPGPU cards without going through the hassle of directly programming the graphics card.

2 Scope of the EASEA Parallelization Platform

Evolutionary algorithms are quite diverse: by 1965, about ten independent beginnings in Australia, United States and Europe have been traced in David Fogel's excellent *fossil record* [12] on evolutionary computing. However, the main evolutionary trends that survived are:

- Evolutionary Programming, by Lawrence Fogel and later David Fogel on the US west coast [11, 13]
- Evolutionary Strategies, by Rechenberg and Schwefel, best described in [4, 21, 22]
- Genetic Algorithms, by Holland, later popularized by Goldberg on the US East Coast (Michigan) [14, 15]
- Genetic Programming, by Cramer [9] and later developed by John Koza [17, 20]

All these algorithms share a common evolutionary loop that was united into the EASEA language, back in 2000 [8], and its Graphic User Interface GUIDE [7] presented in 2003 that is currently being redesigned as part of the French EASEA-CLOUD project.

The first versions of EASEA (until v0.7) produced C++ (or Java) source code for sequential CPUs. Since 2009, EASEA v1.0 can produce massively parallel code for NVIDIA GPGPU cards. Since 2011, v1.1 can implement an island model over several computers connected over the Internet.

Further, the EASEA platform extends into the EASEA-CLOUD platform to run over computing eco-systems, made of heterogeneous resources that can combine the power of CPU or GPU individual computers or Beowulf clusters, or grids, or clouds of computers using an island model to solve the same problem. The web sites for EASEA and EASEA-CLOUD are respectively http://easea.unistra.fr and http://easea-cloud.unistra.fr. Both platforms can be found under Sourceforge (http://www.sourceforge.net).

3 Standard CPU Algorithm

An EASEA (EAsy Specification of Evolutionary Algorithms) compiler was designed to build a complete evolutionary algorithm around four problem-specific functions and a representation of a potential solution all integrated into a `.ez` source file. Throughout this section, an example will be used that implements an evolutionary algorithm whose aim is to find the minimum of a 100-dimensional Weierstrass benchmark function.

Structure of an Individual

Defining the structure of an individual is done in a section devoted to the declaration of user classes that all `.ez` source files must contain. Declarations use a C-like syntax, allowing one to compose a genome out of integers, floats, doubles, and booleans but also pointers:

```
\User classes :
  GenomeClass {
    float x[100];
  }
\end
```

`GenomeClass` is the reserved name for the class that describes individuals. More than one class can be defined, but at least one must be named `GenomeClass`.

Initialization Function

This function tells how to initialize a new individual. It is a simple loop that assigns a random value to all 100 variables of the genome, within X_MIN and X_MAX, which are macro-definitions declared by the user in a user declaration section.

```
\GenomeClass::initialiser :
  for(int i=0; i<100; i++ ) Genome.x[i] = random(X_MIN,X_MAX);
\end
```

`Genome` is the reserved name for the individual to be initialized. `random` is a function of the EASEA library that calls an appropriate random number generator depending on the version of EASEA (typically a Mersenne Twister).

Evaluation Function

A Weierstrass function is used for this example. It is mathematically defined as:

$$W_{b,h}(x) = \sum_{k=1}^{\infty} b^{-kh} \sin(b^{kx}) \quad \text{with } b > 1 \text{ and } 0 < h < 1$$

In this example, we try to minimize the absolute value of this function, hence the evaluation code below for each of the 100 dimensions of the genome. Note that the function is an infinite sum of sines. For practical reasons, only 125 sums are performed.

```
\GenomeClass::evaluator :
  float Res=0, Sum, b=2, h=0.5;

  for (int i=0; i<100; i++) {
    Sum=0;
    for (int k=0; k<125; k++)
      Sum+=pow(b,-(float)k*h)*sin(pow(b,(float)k)*Genome.x[i]);
    Res += (Sum < 0 ? -Sum : Sum);
  }
  return (Res);
\end
```

The evaluation function must return a floating point value that represents the fitness of the individual (note that if one wants to use a roulette-wheel selector as defined in [15], the returned fitness value must be $>=$ 0 and the goal of the evolutionary algorithm must be to maximize the fitness function, which is not the case here).

Crossover Function

In this example, a very basic barycentric crossover is presented :

```
\GenomeClass::crossover :
  for (int i=0; i<100; i++) {
    float alpha = random(0.,1.); // barycentric crossover
    child.x[i] = alpha*parent1.x[i] + (1.-alpha)*parent2.x[i];
  }
\end
```

By default, the crossover function already knows about three individuals: `parent1, parent2, and child`, which are members of the GenomeClass that was defined by the user. By default, `child` is instantiated with a deep copy of `parent1`.

Mutation Function

The very simple mutation function below simply adds a random value in $[-.1, .1]$ and makes sure that x remains within bounds.

```
\GenomeClass::mutator : // Must return the number of mutations
  int nNbMut=0;
  for (int i=0; i<100; i++)
    if (tossCoin(.01)){
      nNbMut++;
      Genome.x[i]+=random(-.01,.01);
      if (Genome.x[i] <= X_MIN) Genome.x[i]=X_MIN;
      if (Genome.x[i] >= X_MAX) Genome.x[i]=X_MAX;
    }
  return nNbMut;
\end
```

In this version, the mutation function walks through all dimensions and mutates each gene with a probability .01, meaning that the probability that the individual will be mutated is 1, because there are 100 dimensions to this problem. The number of mutations must be returned because in some versions of EASEA, this is used for statistics.

Set of Parameters

The task of the EASEA compiler is then to wrap a complete evolutionary algorithm around these problem-specific functions and individual structure definition that can virtually implement any standard or nonstandard evolution engine thanks to a large enough set of parameters and a library of functions.

The typical set of parameters is:

- *Number of generations*: the number of generations for the run to complete.
- *Time limit*: the number of seconds before the run is stopped (0 to cancel this functionality).
- *Population size*: the number of individuals in the population.
- *Offspring size*: the number of children to be created per generation.
- *Mutation probability*: the probability to call the mutation function once a child has been created.
- *Crossover probability*: the probability to call the crossover function to create a child (if the crossover function is not called, the child is a clone of the first parent).
- *Evaluator goal*: to specify whether the evolutionary engine should minimize or maximize the fitness of the individuals.
- *Selection operator*: to be chosen among (Deterministic, Random, Tournament, Roulette). "Roulette" can only be specified if the evaluator goal is "maximize", and Tournament needs a parameter to specify selection pressure. An n-ary tournament will be implemented for integer values of n greater than 2, and a stochastic tournament will be implemented for real values between .5 and 1.0.

(In a stochastic tournament, the best of two individuals is returned with a probability between 0.5 (random selection) and 1.0 (binary tournament).)

- *Surviving parents*: number of parents to participate in the selection of the individuals that will constitute the next generation.
- *Surviving offspring*: number of children to participate in the selection of the individuals that will constitute the next generation.
- *Reduce parents operator*: how the surviving parents are selected.
- *Reduce offspring operator*: how the surviving children are selected.
- *Final reduce operator*: how the next generation is selected among the surviving parents and offspring.
- *Elitism*: strong (meaning that the elite is selected among the parents) or weak (meaning that the elite is selected among the surviving parents + surviving offspring).
- *Elite*: Elite size (number of best individuals to make it to the next generation).

This rather complete set of parameters allows one to specify virtually any evolution engine, from a generational GA (offspring size equal to population size, 0 surviving parents, 100 % surviving offspring, deterministic selectors) to a steady state GA (offspring size equal to 1, weak elitism) to an evolutionary strategy comma (offspring size greater than population size, 0-surviving parents) or an evolutionary strategy plus (surviving parents + surviving offspring > population size).

Note that strong elitism is used with generational algorithms (generational GA or ES-comma) where the elite is chosen among the parents, while non-generational algorithms such as ES-plus can choose their elite among the offspring or parents (weak elitism).

Then, when the `easea` compiler is invoked on an `.ez` file containing the above-mentioned functions and parameters, a man-made template that contains a complete evolutionary algorithm is specialized with the provided user functions and parameters.

The result is a human-readable indented and commented C++ source file that implements a complete algorithm that evolves solutions to the problem that was specified in the evaluation function, using genetic operators specified by the user.

4 Partial and Full Parallelization on One Machine

Evolutionary algorithms are intrinsically parallel, so in theory, all the different steps are fully parallelizable, meaning that Amdahl's law [3] should not hinder speedup (cf. chapter "Why GPGPUs for Evolutionary Computation?").

However, this is only true for a non-elitist generational GA, where the offspring population replaces the parent population: if one wants to implement an elitist generational GA, then at some point, it will be necessary to find out who was the best parent, to copy it into the next generation.

Unfortunately, finding which individual is the best is not intrinsically parallel, because all different cores must then communicate at some point. This implies some synchronization between the cores as well as exchanging information that will degrade speedup.

Worse: if one wants to implement an evolutionary strategy plus (ES+) with a population of 10,240 parents + 10,240 offspring on a 1,024-core GPGPU, it will be necessary to select 10,240 individuals out of 20,480 individuals to create the next generation. Supposing each of the 1,024 cores was asked to select 10 good individuals out of the 20,480-individual temporary population made of parents + offspring to constitute the next generation of 10,240 individuals, it is extremely probable that some good individuals will be chosen many times. *This means that the new generation of 10,240 individuals will contain many clones which will lead to premature convergence.*

One solution is to implement an intrinsically parallel selector without replacement, as was done with the DISPAR tournament [19], but this selector is not absolutely identical to a tournament selection, which means that the massively parallel algorithm will not be strictly equivalent to its sequential counterpart.

Another solution is to only parallelize the evaluation function, which makes sense if its computation is intensive enough to dwarf the time devoted to execute the evolution engine. Using this solution means that both the sequential and parallel algorithms will be strictly identical.

Below is a test done on the Weierstrass function shown above, on a PC with an Intel Core I7 990X running at 3.46 GHz for an advertised computing power of 107 GFlops with an NVIDIA GTX680 with 1,536 cores running at 1 GHz for an advertised computing power of 3.09 TFlops.

For the test below, the exact code above was inserted into an `.ez` file as well as the parameters below that implement a generational GA.

```
\Default run parameters :
  Number of generations : 10000
  Time limit: 0                 // in seconds, 0 to deselect
  Population size : 16384
  Offspring size :  16384
  Mutation probability : 1
  Crossover probability : 1
  Evaluator goal : minimize
  Selection operator: tournament 7
  Surviving parents: 0
  Surviving offspring: 100%
  Reduce parents operator: deterministic
  Reduce offspring operator: deterministic
  Final reduce operator: tournament 2

  Elitism: Strong
  Elite: 0
  ...
\end
```

Compiling the file (under LINUX) with EASEA for CPU (default mode) is done with the following lines:

```
$ easea weierstrass.ez
$ make
...
$
```

Then, execution gives:

```
$ ./weierstrass --nbGen=10
...
Population initialisation (Generation 0)...
GEN      TIME      EVAL     BEST       AVG        STDDEV     WORST
0       85.83     16384   1.11e+02   1.25e+02   3.50e+00   1.39e+02
1      171.44     32768   1.01e+02   1.16e+02   3.35e+00   1.30e+02
2      256.83     49152   9.72e+01   1.09e+02   3.26e+00   1.22e+02
3      342.04     65536   8.81e+01   1.03e+02   3.25e+00   1.16e+02
4      427.01     81920   8.32e+01   9.53e+01   3.15e+00   1.07e+02
5      511.76     98304   7.53e+01   8.73e+01   2.95e+00   9.83e+01
6      596.32    114688   6.77e+01   7.95e+01   2.78e+00   9.17e+01
7      680.66    131072   5.98e+01   7.18e+01   2.57e+00   8.24e+01
8      764.81    147456   5.42e+01   6.46e+01   2.38e+00   7.31e+01
9      848.77    163840   4.72e+01   5.78e+01   2.15e+00   6.58e+01
Current generation 10 Generational limit : 10
Stopping criterion reached
```

Note that the command line parameter `--nbGen=10` overrides the default parameter that was in the `.ez` file that was saying:

```
Number of generations : 10000
```

One can see that each generation that evaluates 16,384 individuals takes around 85 s to compute, meaning that one evaluation takes less than 5.24×10^{-3} s, because in each generation the time taken by the evolution engine is also included.

After 10 generations and 163,840 evaluations (remember that the algorithm presented above is generational), the value found for the best individual is 47.2.

Now, using the very same `weierstrass.ez` source file, EASEA can wrap around its contents a complete algorithm that performs a massively parallel evaluation on the NVIDIA GPGPU card, if its compiler is invoked with the `-cuda` command line option. Execution time is then different:

```
$ easea weierstrass.ez -cuda
$ make
...
$ ./weierstrass --timeLimit=849
...
Population initialisation (Generation 0)...
card (0) GeForce GTX 680, 16384 individuals: t=1024 b: 16
GEN      TIME      EVAL     BEST       AVG        STDDEV     WORST
0         0.16    16384   1.09e+02   1.25e+02   3.49e+00   1.39e+02
1         0.41    32768   1.04e+02   1.16e+02   3.32e+00   1.28e+02
2         0.66    49152   9.68e+01   1.10e+02   3.24e+00   1.23e+02
3         0.90    65536   9.02e+01   1.03e+02   3.23e+00   1.15e+02
```

```
4       1.14     81920  8.38e+01  9.55e+01  3.13e+00  1.07e+02
5       1.38     98304  7.60e+01  8.75e+01  2.97e+00  1.01e+02
6       1.62    114688  6.92e+01  7.97e+01  2.75e+00  9.12e+01
7       1.86    131072  6.23e+01  7.20e+01  2.56e+00  8.20e+01
8       2.11    147456  5.59e+01  6.48e+01  2.36e+00  7.45e+01
9       2.35    163840  4.90e+01  5.80e+01  2.14e+00  6.59e+01
...
```

On this quite fast evaluation function (less than .005 s on a CPU), it now only takes 2.35 s to perform the same ten generations, meaning that the obtained speedup is around 360×.

Results are different because the algorithm is stochastic: random choices are made when selecting parents, so results are different between executions.

Compiling with the `-cuda` option means that the EASEA platform compiled the evaluation function with the `nvcc` NVIDIA C Compiler. Then, whenever an evaluation phase occurs, the complete population is collected in one chunk of memory, then transferred in one go onto the GPGPU card and evaluated in parallel on the cores of the card.

All this was done thanks to the `-cuda` command line option of the EASEA compiler. The fact that the `weierstrass.ez` file is identical guarantees that the parallel algorithm is identical to the sequential one.

Because the CPU version evolved the ten generations in 848.77 s, we ran the evolutionary algorithm for 849 s (`--timeLimit=849` parameter on the command line) to find out what result would have been found in the same amount of time as the sequential CPU algorithm:

```
3621  845.99  59342848  6.19e-01  8.58e-01  2.38e-01  2.47e+00
3622  846.22  59359232  6.19e-01  8.58e-01  2.35e-01  2.56e+00
3623  846.46  59375616  6.18e-01  8.61e-01  2.44e-01  2.56e+00
3624  846.69  59392000  6.18e-01  8.59e-01  2.40e-01  2.23e+00
3625  846.92  59408384  6.17e-01  8.56e-01  2.37e-01  2.19e+00
3626  847.15  59424768  6.16e-01  8.58e-01  2.38e-01  2.32e+00
3627  847.39  59441152  6.16e-01  8.56e-01  2.36e-01  2.22e+00
3628  847.62  59457536  6.16e-01  8.59e-01  2.39e-01  2.42e+00
3629  847.85  59473920  6.16e-01  8.55e-01  2.36e-01  2.09e+00
3630  848.90  59490304  6.16e-01  8.53e-01  2.38e-01  2.16e+00
Time Over
Time Limit was 849 seconds
```

Because evolutionary computing relies on evaluations to evolve new solutions through genetic operators, Fig. 1 shows that having access to more computing power will allow better solutions to be found in the same amount of time.

One can see that 3,630 generations were evolved thanks to the GPGPU parallelization while only 10 generations were obtained for the sequential algorithm on one of the cores of the CPU. This confirms the 360× speedup that was computed on the difference of execution time between CPU and GPGPU versions, meaning that 10 s on the GPU is equivalent to 1 h execution on the CPU.

Obtaining a value of .616 would therefore have taken 85 h (3.5 days) on the CPU, and 1 day computation on the GPU is equivalent to 1 year (360 days) on the CPU!

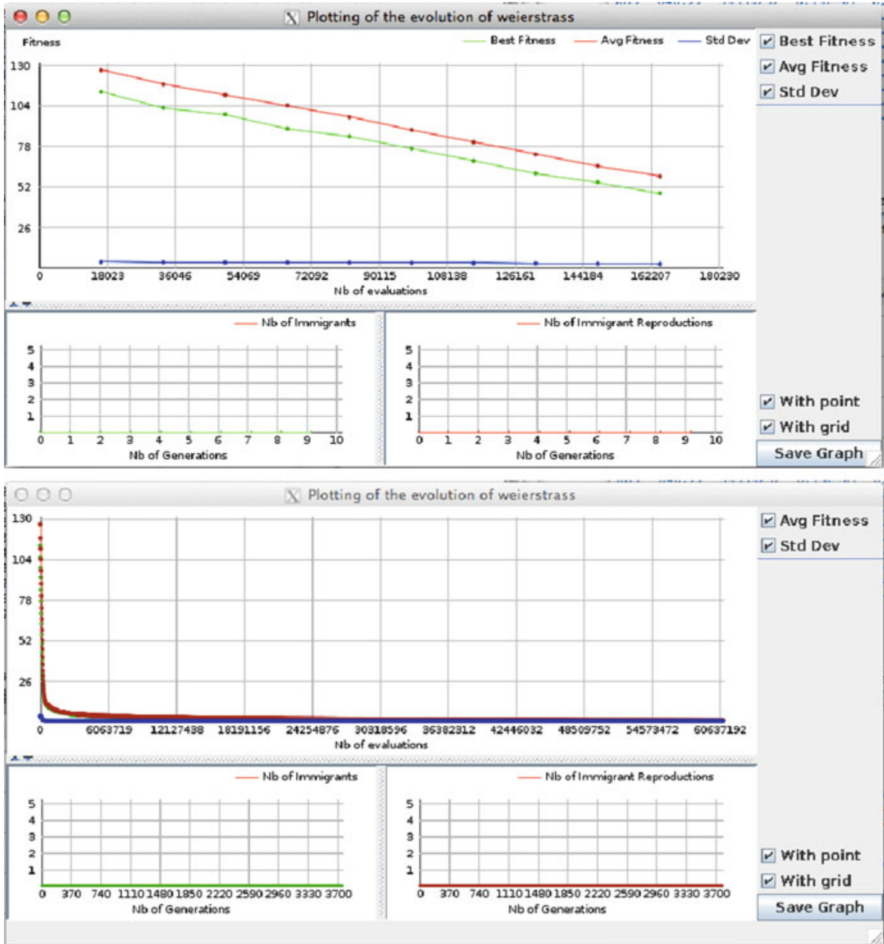

Fig. 1 The EASEA platform provides a graphic display that shows (among other information) the fitness of the best individual, the average of the population and the standard deviation. The *top* and *bottom* figures show the obtained results for the *same* algorithm on CPU and GPGPU, run for the same amount of time (849 s)

The speedup offered by a GPGPU card may allow one to tackle problems that are currently beyond the reach of a sequential algorithm.

Because the parallelization is performed on the evaluation function only (and supposing that parallelization was absolutely perfect over an infinite number of cores), it is important to stress that the obtained speedup depends on how computer intensive is the evaluation function and that for a particular problem, the maximum speedup can only be obtained with population sizes about an order of magnitude greater than the number of cores of the used GPGPU card.

On this example, obtaining a 360× speedup means that the sequential part of the algorithm is 1/360th of the complete algorithm, suggesting that the parallel part of the algorithm represented 99.9972 % of the complete algorithm.

If the EASEA automatic evaluation parallelization is tested on a function that represents less than 99.9972 % (i.e. is shorter to evaluate than .005 s on the CPU), Amdahl's law will apply and the speedup will not be as good as 360×. This is the price to pay for using a strictly identical algorithm to the sequential one.

On a much simpler function, for instance, the obtained speedup can be <1 because of the overhead induced by transferring the evaluation onto the GPGPU card. For instance, the speedup obtained on the following Rosenbrock function:

$$R(x) = \sum_{i=1}^{\dim}[(1 - x_i)^2 + 100(x_{x+1} - x_i^2)^2] \tag{1}$$

shown in Fig. 2 top can be as low as 0.65× on an NVIDIA GTX480 card vs. one core of an Intel Core i7 950 CPU running at 3.07 GHz, for 80 dimensions and 10,000 individuals.

In this case, in order to fully exploit the GPGPU card, it is necessary to use a fully parallel implementation on a GPU, using either a parallel simulation of a tournament for the reduction phase (DISPAR tournament [19] in Fig. 2 bottom left) or a generational algorithm (in Fig. 2 bottom right) on which accelerations up to 250× can be obtained even on this very lightweight evaluation function, on a GTX480 card `vs.` one core of a good CPU (Intel Core I7 950) of the same generation.

In order to better understand what happens when parallelizing EAs on GPGPU cards, Fig. 3 shows the time distribution between the different phases.

The top line (ezCPU) shows (from left to right):

1. The initialization phase (nearly invisible)
2. The evaluation phase
3. The parents selection phase (tournament in this case)
4. The variation phase (creation of children using crossover and mutation) and
5. The reduction phase (reducing the parents + children population to create the next generation)

on the same Weierstrass function with parameters tuned so that the disproportion between the evaluation phase and all the other phases is not too accented.

The line below (ezGPU) shows what happens when the same program is compiled by EASEA using the `-cuda` option. The evaluation phase nearly disappears as the evaluation of the 4,800 individuals is parallelized over the 480 cores of the NVIDIA GTX480. What remains is the sequential part of the algorithm, showing Amdahl's law at work.

The only way to obtain a good speedup in this configuration is to parallelize all the phases of the algorithm, which results in the two lines below (gpuDispar and gpuGen), in which the distribution of the different phases is indiscernible.

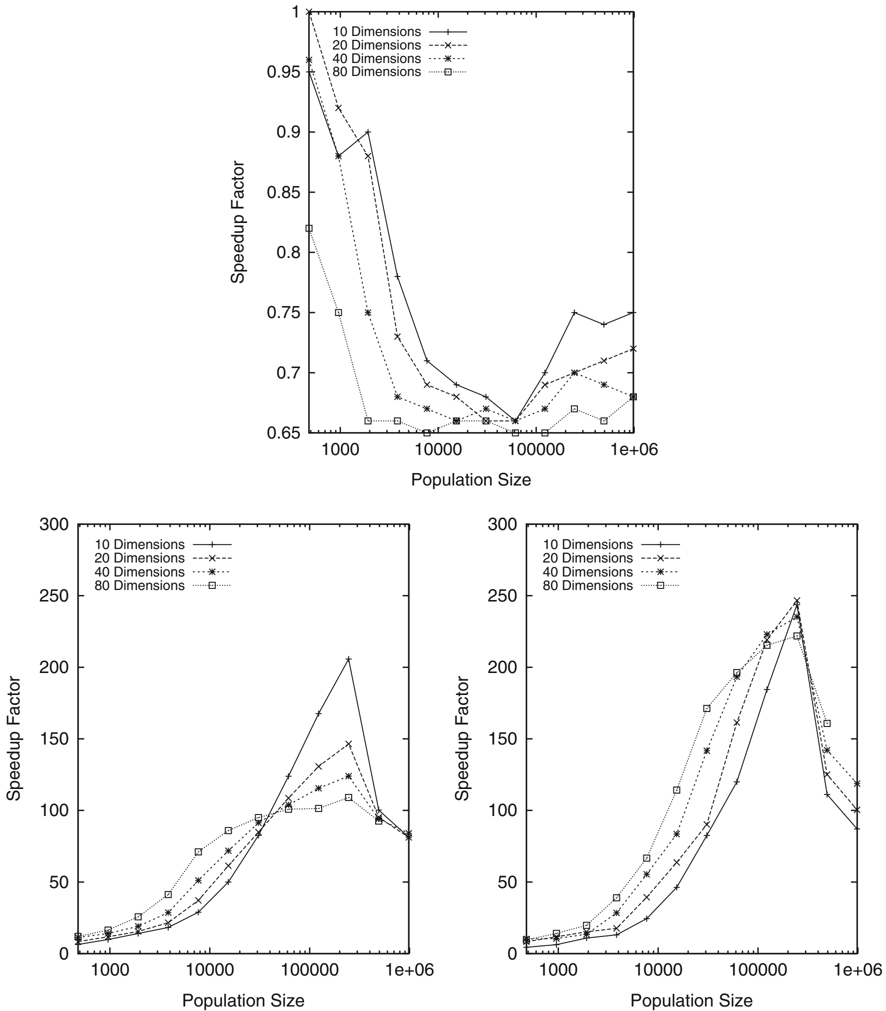

Fig. 2 The *top* figure shows the obtained speedup with a standard EASEA parallelization on the lightweight Rosenbrock function on an NVIDIA GTX480 card for 10–80 dimensions. The *bottom left* figure is obtained with a fully parallel EA using a DISPAR tournament and the *bottom right* figure with a generational fully parallel GA

In order to see something, Fig. 4 shows the same lines as a percentage of the total time. As above, the first (ezCPU) serves as a reference before parallelization. The second line shows the much smaller proportion of the evaluation phase.

The third and fourth lines show completely parallelized algorithms, so if all phases are equally parallelized, the proportions should be similar to those of the first line.

On the third line, the DISPAR tournament used to reduce the population takes a bit more time than the standard tournament, and the initialization phase now

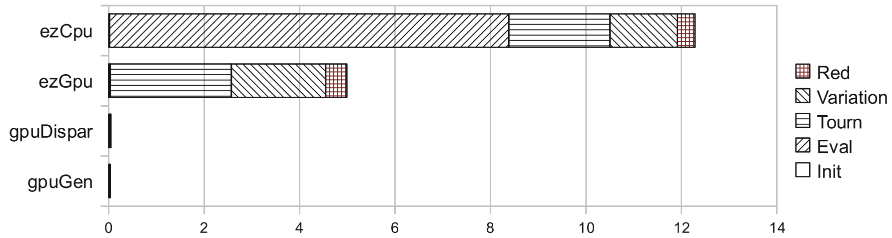

Fig. 3 Time distribution on a GTX480 NVidia card and i7 950 CPU for Weierstrass function with 4,800 individuals, 10 iterations and 50 generations (in seconds), over an average of 20 runs

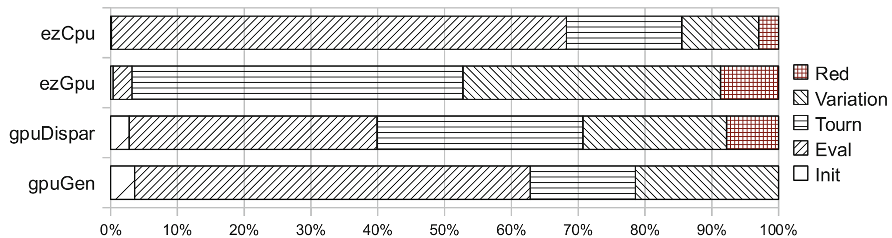

Fig. 4 Same as Fig. 3, but as a percentage of execution time

appears. On the fourth line, the generational replacement uses no time, and the proportions are similar to those of the CPU algorithm. An average was taken over 20 runs because on such small execution times for the parallel version, even small variations will change proportions.

Full parallelization of an evolutionary algorithm using a DISPAR or generational reduction phase will soon be available under EASEA.

5 Island Parallelization and Emergent Behaviour

As was said in the chapter "Why GPGPUs for Evolutionary Computation?", driving through a desert sea of sand dunes can be very tricky, as it is very easy for a vehicle to get stranded in soft sand. In terms of search algorithms, one can say that an evolutionary algorithm can get stranded in a multimodal search space if its population has prematurely converged to a local optimum.

Unfortunately, the task of search algorithms is even more difficult than that of crossing a desert of soft sand dunes because their task is to explore the pits to find the deepest ones. While doing so, the chances that algorithms get stranded (read "premature convergence") are much larger than if they could drive around the ditches.

However, desert drivers found that a complex system was much more efficient to cross the desert. Complex systems can be seen as a group of autonomous entities in

Fig. 5 4WD vehicles implementing an emergent complex system to cross the desert

interaction that implement an emergent behaviour. In Aristotle's terms, the whole is more than the sum of the parts.

Rather than attempting to cross the desert alone, 4WD drivers use several vehicles (cf. Fig. 5). If there were no interaction between the vehicles, the result would be the same as if they used the same vehicle n times: each vehicle would eventually get stranded with the same probability as the other vehicles.

What turns this group of vehicles into a complex system with emergent behaviour is that they carry tow cables and winches that allow them to interact. Indeed, if a vehicle is stranded into a pit, another one can throw it a cable and pull it out of the pit with its winch. The vehicle is not stranded anymore and the group can resume their progression across the desert.

In this case, the emergent behaviour is that where n vehicles will get stranded as easily as one vehicle, n vehicles with tow cables and winches will be able to cross the desert. The whole is more than the sum of the parts.

Evolutionary algorithms already exploit a similar behaviour because they are population-based algorithms. Where one could use n independent simulated annealing algorithms, evolutionary algorithms are more efficient because they use n individuals in interaction (through crossover) to help themselves out of local optima and preserve diversity.

However, even with a large population, algorithms will eventually get stranded as their population will converge into a sweet spot. One can then implement a higher-level kind of parallelism, by getting several evolutionary algorithms to cooperate on the same search space, thereby implementing a multi-level complex system: an island-model evolutionary algorithm [1].

If several independent population-based algorithms (let us call them islands) explore the same search space, they can act like several vehicles exploring an erg and help one another out of local optima in order to find even better areas.

If they get stranded (cf. Fig. 6 left), rather than using a tow cable, an EA island can send its best individual to another island (cf. Fig. 6 right). The individual is not physically sent to the location where the other island resides. The individual is made

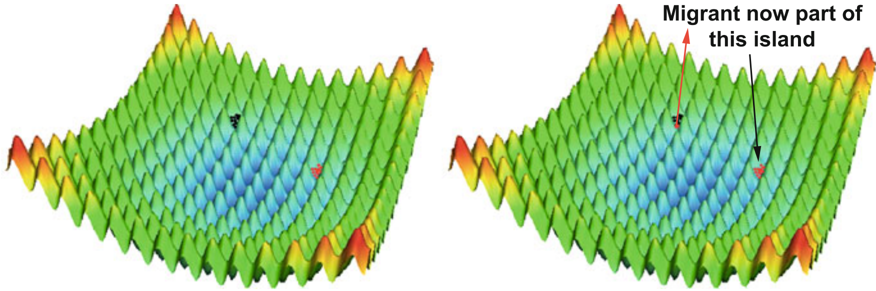

Fig. 6 On this Rastrigin function, two islands have converged towards local optima (*left*). Then, an individual is sent from the left island to the right island. Note that the individual does not physically move: it stays where it is but it is now part of the population of the right island

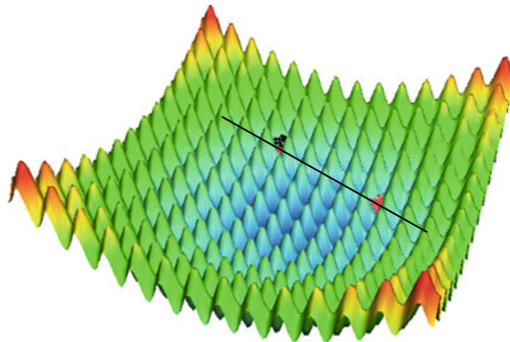

Fig. 7 If the migrant is better than the local individuals of the right island, a barycentric crossover (such as BLX-α with $\alpha = .5$) will distribute offspring across the line. The island, which had prematurely converged, will be pulled out of its local optimum by the immigrant

part of the population of the other island, but it stays topologically where it was in the search space.

If the incoming individual (immigrant) is worse than the individuals in the stranded island, it will get discarded and will not be part of the next generation. If however, it fares well among the population of the island where it was sent or if it is better than the best individuals of the island, this immigrant will implement a tow cable between the two islands:

- It will survive across the generations.
- Other individuals will want to mate with it to create new offspring.

If a barycentric-type crossover is used (a BLX-α [10] with $\alpha = .5$ as in Fig. 7 for instance), some created offspring will keep appearing on a line joining the two islands until a better solution than the one implemented by the migrant is found.

If n stranded islands share the same search space and exchange individuals, children will spawn over $n(n-1)/2$ lines, therefore improving the chances to find

better places in the search space, as individuals situated on the lines may also want to create children.

However, individuals should not be exchanged among islands too often, or the multi-island algorithm will become panmictic (with a global population) across the different islands. Ideally, local exploitation should take place in between migrations so that new good spots are well explored before the island is pulled out of its new local optimum by another good migrant.

Typically, search algorithms must stay on the ideal EvE (Exploration vs. Exploitation) compromise. In evolutionary algorithms, this is done by fighting against premature convergence while allowing the algorithm to explore enough local optima.

In an island model, one can implement fast-converging islands (something easy to do because evolutionary algorithms can be made quite convergent if strong selection pressure operators are chosen) that will be periodically pulled out of their local optima by good incoming immigrants.

Needing islands to find the bottom of local optima before they are pulled out of them means that infrequent communication is actually an advantage over frequent communication! This is a very important point, as communication is usually what prevents parallel machines from yielding linear speedup with the number of machines: usually, 10 machines will not go $10\times$ faster than one machine because of the needed synchronization and communication time between machines. Super-fast communication networks between machines are one of the primary costs of supercomputers! Being able to get computers to cooperate over a loosely coupled asynchronous low-bandwidth network means that it is possible to create supercomputers out of standard machines that are cooperating together over the Internet, for instance.

In an island-model parallelization scheme, communication (migration) will actually benefit from taking place asynchronously and in a loosely coupled manner!

A lot of research has already been done on island models. Theoretical studies on the number and the size of populations have been done by Whitley et al. [24]. Other studies show the influence of different island-model parameters such as migration rate and connectivity [6]. Alba and Troya studied asynchronism [2]; Branke et al. studied the influence of heterogeneous networks on island models [5].

Some research has been done on parallelizing evolutionary computation using an island model on a GPGPU card, but this research was limited to a single machine equipped with a GPGPU card that hosted all the islands [23].

Some results obtained with the EASEA island model have been quoted in the Supporting Online Material of a Science paper [16] whose results were obtained with the help of the work described in the chapter "Using Large-Scale Parallel Systems for Complex Crystallographic Problems in Materials Science".

5.1 EASEA Island Model

The default island model implemented by the EASEA language is very basic but very versatile at the same time. It allows the algorithm to periodically send (and receive) one or several individuals to (from) one of several IP addresses and ports listed in a file.

The configuration of the EASEA island model relies on two main parameters of the `.ez` file:

- The name of the file containing the IP addresses and ports (`ip.txt` by default)
- The migration probability

Here is a basic `ip.txt` file where four islands are implemented on the same local machine and four other islands on a distant machine:

```
$ cat ip.txt
127.0.0.1:2929
127.0.0.1:2930
127.0.0.1:2931
127.0.0.1:2932
130.79.92.66:2929
130.79.92.66:2930
130.79.92.66:2931
130.79.92.66:2932
$
```

This allows one to implement several islands on a single machine (for multi-core machines, with different ports on a single IP number) or on different machines (with different IP numbers).

All the islands can use an identical (or a different) `ip.txt` file as when an individual is sent the island makes sure it does not send the individual to itself.

Individuals are sent between islands using a connectionless UDP/IP protocol (this can be seen as individuals sent by mail between the different islands). No connexion means that the process can be asynchronous and therefore quite robust: an island can stop working (due to a hardware problem for instance) and then start again later on. It will receive good individuals from other machines that did not crash and will rapidly find good spots again without disturbing the other islands.

New islands can join the search at any moment.

Sent individuals may get lost, but then the whole EC island paradigm is stochastic, so losing an individual once in a while can be considered as being part of the algorithm.

No time is lost in island synchronization or acknowledgement that a sent individual has effectively been received.

As EASEA can produce code for machines running with or without GPU, running under Linux, Windows or Mac Os X, the EASEA island model can run on any number of heterogeneous machines connected via the Internet worldwide to work on solving the same problem, provided each IP address is present in the IP file of each island and provided they exchange individuals sharing the same structure.

With the EASEA-CLOUD project, this will be extended to Grid and Cloud computing, for a full massively parallel evolutionary computation eco-system.

Algorithms may be different depending on machines provided that they can share the same individuals: slower machines can run exploratory algorithms, while faster ones can concentrate on exploitation of good spots.

5.2 Implementation

Before the evolution begins on an island, the file containing the IP+port addresses of the other islands is parsed, creating a list of clients. The IP+port address of an island can appear several times. Because one IP+port number is picked at random among the number of IP+port entries, replicating an identical IP+port number ten times will give it ten times more chances to be selected than the other islands, therefore allowing one to implement weighted edges between islands.

Then, communication need not be symmetric, as an island can know the IP+port number of another one, but the second may not have the IP+port number of the first. In a heterogeneous environment (machines with GPUs and machines without GPUs, for instance), this will allow one to implement unidirectional flows of individuals between slow exploratory machines that find good spots and fast exploitation machines that concentrate on finding local minima. However, the fast machines may not send back individuals to the slow machines so as to prevent premature convergence in the cluster of slow machines.

Then, at every generation, a migration function is called with a probability p set by the user. A destination island is chosen randomly among the list of clients. Once a destination is chosen, n individuals are selected among the population using a user-defined selector. The selected individuals are then serialized and sent to the destination island.

On every island, at the beginning of the run, a thread is launched that is in charge of receiving incoming individuals. Upon arrival of a newcomer, a user-defined selector decides who in the population will be replaced. The integration process for newcomers is performed at every new generation.

The described implementation of an island model is very simple, robust and versatile, as it allows one to easily design complex topologies. For instance, it would be really easy to create a ring topology by just giving each island the address of the following island. One could also create a square or hexagonal toroidal grid or virtually any other topology. Then by changing the migration probability, the communication rate can easily be increased or decreased: for instance, the centre island in a star-shaped topology may need more frequent communications with other islands, or one could imagine modifying the migration rate depending on some strategy (increase or decrease migration with the number of generations).

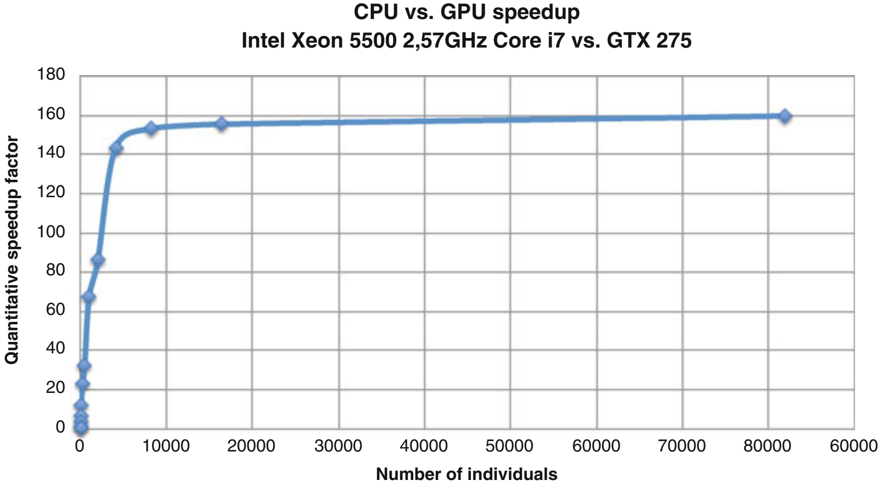

Fig. 8 GPU vs. CPU speedup on a 240-core GTX275 NVIDIA card vs. an Intel XEON 550 CPU of the same generation depending on population size. The Weierstrass benchmark is set up over 1,000 dimensions with 120 iterations and using a Hölder parameter value of 0.35. Greater population sizes give better speedups

6 Experiments

In this section, the EASEA GPU island model is tested with an old cluster (a classroom) of 20 Intel Xeon 5500 Core i7 PCs, each containing a 240-core NVIDIA GTX275 GPU card, for a total computing power of around 20 TFlops.

6.1 Quantitative GPU vs. CPU Speedup

First of all, before the island model is tested, it must be recalled that evaluation can be parallelized on each machine's GPU card, so Fig. 8 presents the obtained speedup by comparing the standard sequential implementation of the Weierstrass problem by the EASEA compiler on one core of an Intel Xeon 5500 2.57 GHz Core i7 CPU with 8 GB Ram vs. the same code compiled with the `-cuda` command line parameter, using an NVIDIA GTX275 card of the same generation with 240 cores.

Different population sizes were used ranging from 8 to 81,920, showing that the obtained speedup reaches a plateau at around 160× for a population size larger than 8,192, while a still-reasonable speedup can be obtained with 4,096 individuals only.

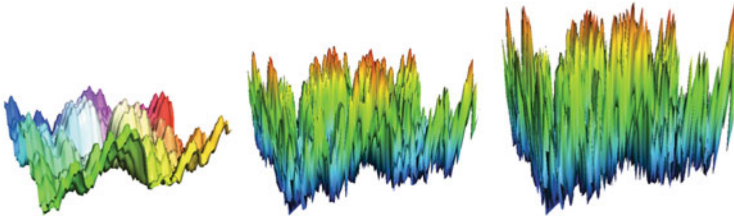

Fig. 9 Weierstrass function for Hölder coefficients 0.9, 0.5 and 0.35 from left to right

6.2 *Qualitative Speedup for the Island Model*

As a benchmark function, we used the same Weierstrass function, whose irregularity can be tuned thanks to its Hölder coefficient h.

Because we are interested in results in this section, we needed to tune the difficulty of the benchmark in order to show the *qualitative* speedup that can be obtained using an island model.

Usually, a Hölder coefficient of 0.5 is used but it turned out that this value created a too-simple function for the 20 TFlop cluster, so irregularity was increased by using a 0.35 Hölder coefficient (irregularity increases as h decreases) and 120 iterations (cf. Fig. 9).

Then two dimensions only was too small a search space for a 20 TFlops cluster, so a 1,000-dimensional problem was used to create a tough enough problem on which conclusions could be drawn.

Knowing that quantitative computing power is exactly linear with the number of machines, it is interesting to have a look at the obtained *quantitative* speedup. How much faster will it be to find the same result on 5, 10 or 20 machines than on one single machine ?

Looking at Fig. 8, it was decided that 4,096 was the best compromise between population size and obtained GPU speedup.

In order to only examine the influence of the island model, all experiments are done using not only identical parameters but also constant population size because population size will play a role in exploration and premature convergence. Indeed, if premature convergence is what an island model is intended to prevent, it would be unfair to compare:

- A single population of 4,096 individuals on one machine for 2,000 generations, to
- 20 populations of 4,096 individuals on 20 machines for 100 generations,

because evolving 81,920 individuals for 100 generations would not converge as fast as 4,096 individuals over 2,000 generations.

So in order to keep things fair, the setup is the following: one population of 81,920 individuals on one machine will be compared to:

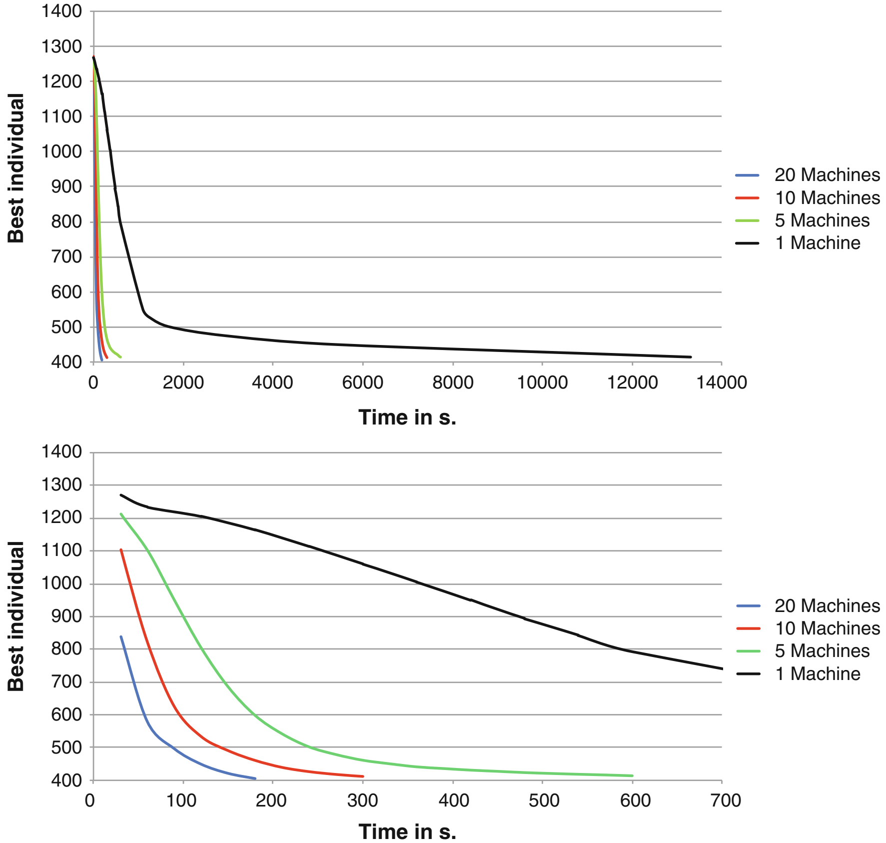

Fig. 10 Evolution of the best individual on different cluster sizes, averaged over 20 runs. The *bottom* figure is a zoom of the *top* figure, so that the different curves for 5, 10 and 20 machines can be seen correctly

- 5 populations of 16,384 individuals on 5 machines
- 10 populations of 8,192 individuals on 10 machines
- 20 populations of 4,096 individuals on 20 machines

The topology used for the experiments is a very straightforward fully connected network (each island uses the same IP file containing the IP numbers of all participating machines, and no machine can send an individual to itself).

Figure 10 top shows four curves. The black curve presents the evolution of the average fitness of the best individual of 20 runs for 14 h on the Weierstrass benchmark using a population of 81,920 individuals. The average best value found over 20 runs was 414.

Then, the three other curves show the average fitness of the best individuals of 20 runs on the same benchmark with 5 machines and 16,384 individuals per machine,

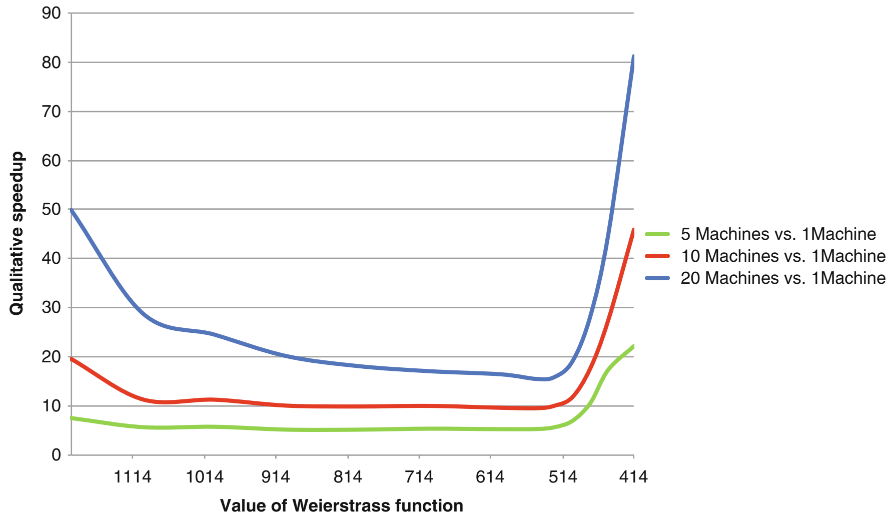

Fig. 11 Qualitative speedup factor determined by comparing one machine vs. 5, 10 and 20 machines in order to reach the same fitness value. These results represent an average of 20 runs

10 machines and 8,192 individuals per machine and finally, 20 machines and 4,096 individuals per machine until the value 414 is reached.

What is interesting to see is that the black curve for one machine shows a marked knee at around fitness 520. The huge 81,920-individual panmictic population quickly gets to this value, after which things get really difficult. This knee very probably shows that the whole population has converged to a local optimum. What happens next is probably only due to a random search implemented by the mutation function (Schwefel adaptive mutation in this algorithm), hence the very slow slope.

On the contrary, no marked knee is seen for the multi-island implementation (cf. Fig. 10 bottom that show the same curves, but over the first 700 seconds only so that the multi-island curves are more visible).

Search becomes more difficult, but no real knee is to be seen, and certainly not around value 500 even though the global population is exactly the same size.

As expected, 20 machines obtain results faster than one machine only, but this figure does not allow one to visualize the obtained speedup well.

In order to better see what is going on, Fig. 11 shows how long it takes each configuration to reach predefined values such as 1,100, 1,000, 900, ..., by horizontally slicing Fig. 10 and looking at how much faster five machines reach these values than one machine and the same for 10 and 20 machines.

It can then be said that Fig. 11 shows *qualitative* speedup curves, i.e. how much faster a particular island configuration finds the same value compared to one panmictic population of the same size on a single machine.

Figure 11 can then be read the following way: it was about six times faster for a five-island configuration (one island per machine) to obtain a fitness of

1114, compared to one machine only, meaning that a slightly super-linear speedup is obtained for five machines. Indeed, five machines have exactly five times the computing power of one machine, so a perfectly linear speedup should have shown an exact 5× speedup.

After value 1114, five islands on five machines show a relatively constant speedup of 5× until value 520 is reached, after which qualitative speedup increases up to more than 20 for value 414 (these are all average values over 20 runs).

The speedup factor for a 10-machine cluster is quite similar: after a good start, speedup stabilizes at around 10× up to value 520 after which speedup rises to around 45×.

With 20 machines, speedup starts with a super-linear speedup of around 50× before going down to nearly 15× and rising again at around value 520, to slightly above 80×.

What probably happens here is that speedup is roughly linear with the number of machines until value 520, after which single machines with one panmictic population of 81,920 individuals get stranded in a sand dune pit. At this point, multi-island models show their advantage as they help each other out of local optima and continue their progression.

Because the speedup keeps rising after value 520, we call it a supra-linear speedup because it rises steeply.

Twenty machines get down to values around 300 in around 10 min, whereas it could take days or months for one machine to get there using random search only.

Interestingly enough, one sees an inflection on the five-machine configuration around value 450, probably because the five islands are getting stranded too, which is not the case for the 10- and 20-island configurations.

Finally, one can see that beyond value 500 (i.e. after single islands tend to get stuck in local optima), the speedup between multi-island configurations remains roughly the same: for value 414, the speedup of a five-machine cluster is slightly above 20×, while it is roughly 45× for 10 machines and above 80× for 20 machines, which is quite satisfying. The reader must be reminded that these curves are for GPU-islands for which Fig. 8 shows that they are already 160× faster than a sequential execution of the same code on a CPU. Therefore, during the linear speedup phase, the speedup of the 20 GPU cluster vs. one single machine is of $\sim 160 \times 20 = 3{,}200$, reaching $\sim 160 \times 80 = 12{,}800$ when single machines get stranded. On this problem, a one-day run on this cluster would therefore be equivalent to at least 35 years on a single island but probably much more as value 414 was obtained in less than 3 min only, and the speedup slope is quite steep.

This island-model speedup helped on the real-world problem of zeolite structure determination described in the chapter "Using Large-Scale Parallel Systems for Complex Crystallographic Problems in Materials Science", where 11 years of calculation were reduced to 3 days on the cluster of 20 machines described above (not taking supra-linear acceleration into account).

7 Conclusion

The EASEA platform is an easy way to test massive parallelization of evolutionary algorithms over one or several GPGPU cards and possibly over several homogeneous or heterogeneous computers equipped (or not) with such cards. The EASEA-CLOUD project will extend this to Grid computing and to the Cloud, so that, in the end, it is possible to use a supercomputer such as TITAN or a computational ecosystem to solve a single large problem.

EASEA is open source and is accessible through its web page: http://easea.unistra.fr.

It produces human-readable code, meaning that the EASEA platform can be used as a primer, to start on a massively parallel implementation (implement a basic program that implements your problem and then modify the C++ code to exactly suit your needs) or to code a complete project from A to Z.

EZ source files are portable, meaning that you can try them for yourself and exchange files on collaborative projects. This is why EASEA was included in the tutorial part of this book and why several chapters are based on it.

References

1. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. Evol. Comput. **6**(5), 443–462 (2002)
2. Alba, E., Troya, J.: An analysis of synchronous and asynchronous parallel distributed genetic algorithms with structured and panmictic islands. In: Rolim, J., Mueller, F., Zomaya, A., Ercal, F., Olariu, S., Ravindran, B., Gustafsson, J., Takada, H., Olsson, R., Kale, L., Beckman, P., Haines, M., ElGindy, H., Caromel, D., Chaumette, S., Fox, G., Pan, Y., Li, K., Yang, T., Chiola, G., Conte, G., Mancini, L., Mery, D., Sanders, B., Bhatt, D., Prasanna, V. (eds.) Parallel and Distributed Processing. Lecture Notes in Computer Science, vol. 1586, pp. 248–256. Springer, Berlin (1999)
3. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, pp. 483–485. ACM, New York (1967)
4. Beyer, H.G., Schwefel, H.P.: Evolution strategies: a comprehensive introduction. Nat. Comput.: Int. J. **1**(1):3–52 (2002)
5. Branke, J., Kamper, A., Schmeck, H.: Distribution of evolutionary algorithms in heterogeneous networks. In: Genetic and Evolutionary Computation? GECCO 2004. Lecture Notes in Computer Science, vol. 3102, pp. 923–934. Springer, Berlin (2004)
6. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell (2000)
7. Collet, P., Schoenauer, M.: GUIDE: unifying evolutionary engines through a graphical user interface. In: Liardet, P., et al. (eds.) EA'03, Marseilles. Lecture Notes in Computer Science, vol. 2936, pp. 203–215. Springer, Berlin (2003)
8. Collet, P., Lutton, E., Schoenauer, M., Louchet, J.: Take it EASEA. In: Schoenauer, M., et al. (ed.): Proceedings of the 6th Conference on Parallel Problems Solving from Nature, LNCS 1917, pp. 891–901. Springer, Berlin (2000). http://sourceforge.net/projects/easea
9. Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In: Proceedings of an International Conference on Genetic Algorithms and their Applications, pp. 183–187 (1985)

10. Eshelman, L., Schaffer, J.D.: Real-coded genetic algorithms and interval-schemata. In: Whitley, L.D. (ed.) Foundations of Genetic Algorithms 2, pp. 187–202. Morgan Kaufmann, Los Altos (1993)
11. Fogel, D.B.: An analysis of evolutionary programming. In: Fogel, D.B., Atmar, W. (eds.) Proceedings of the 1st Annual Conference on Evolutionary Programming, pp. 43–51. Evolutionary Programming Society, La Jolla (1992)
12. Fogel, D.B.: Evolutionary Computing: The Fossil Record. IEEE Press, Los Alamitos (1998)
13. Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial Intelligence Through Simulated Evolution. Wiley, New York (1966)
14. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading (1989)
15. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
16. Jiang, J., Jorda, J.L., Yu, J., Baumes, L.A., Mugnaioli, E., Diaz-Cabanas, M.J., Kolb, U., Corma, A.: Synthesis and structure determination of the hierarchical meso-microporous zeolite itq-43. Science **333**(6046), 1131–1134 (2011)
17. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Evolution. MIT Press, Cambridge (1992)
18. Maitre, O., Kruger, F., Querry, S., Lachiche, N., Collet, P.: Easea: specification and execution of evolutionary algorithms on GPGPU. J. Soft Comput. **16**(2), 261–179 (2012)
19. Maitre, O., Lachiche, N., Collet, P.: Two ports of a full evolutionary algorithm onto GPGPU. In: Hao, J.K., Legrand, P., Collet, P., Monmarche, N., Lutton, E., Schoenauer, M. (eds.) Artificial Evolution. Lecture Notes in Computer Science, vol. 7401, pp. 97–108. Springer, Berlin (2012)
20. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. In: Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (With contributions by J. R. Koza) (2008)
21. Rechenberg, I.: Evolutionstrategie: Optimierung technischer Systeme nach Prinzipien des biologischen Evolution. Frommann-Holzboog Verlag, Stuttgart (1973)
22. Schwefel, H.P.: Numerical Optimization of Computer Models. Wiley, New York (1981) [1995—2nd edn.]
23. Van Luong, T., Melab, N., Talbi, E.-G.: GPU-based Island Model for Evolutionary Algorithms. In: Genetic and Evolutionary Computation Conference (GECCO), Portland, USA (2010)
24. Whitley, D., Rana, S., Heckendorn, R.B.: The island model genetic algorithm: on separability, population size and convergence. J. Comput. Inform. Technol. **7**, 33–48 (1999)