J. Parallel Distrib. Comput. ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Solving very large instances of the scheduling of independent tasks problem on the GPU

Frédéric Pinel[a], Bernabé Dorronsoro[b,*], Pascal Bouvry[a]

[a] *Faculty of Science, Technology, Communications, University of Luxembourg, Luxembourg*

[b] *Interdisciplinary Centre for Security, Reliability, and Trust, University of Luxembourg, Luxembourg*

ARTICLE INFO

Article history:
Received 3 June 2011
Received in revised form
23 February 2012
Accepted 27 February 2012
Available online xxxx

Keywords:
Cellular genetic algorithm
GPU
Scheduling

ABSTRACT

In this paper, we present two new parallel algorithms to solve large instances of the scheduling of independent tasks problem. First, we describe a parallel version of the Min–min heuristic. Second, we present GraphCell, an advanced parallel cellular genetic algorithm (CGA) for the GPU. Two new generic recombination operators that take advantage of the massive parallelism of the GPU are proposed for GraphCell. A speedup study shows the high performance of the parallel Min–min algorithm in the GPU versus several CPU versions of the algorithm (both sequential and parallel using multiple threads). GraphCell improves state-of-the-art solutions, especially for larger problems, and it provides an alternative to our GPU Min–min heuristic when more accurate solutions are needed, at the expense of an increased runtime.

1. Introduction

Task scheduling reflects some of the most important issues in clusters. The goal is to execute all the arriving tasks at minimum cost. This cost could be measured, for instance, in terms of the time needed to compute the tasks, the use of resources, the energy consumed, or any combination of them.

We focus in this paper on minimizing makespan for the batch scheduling of independent tasks on a fixed number of machines. We assume that the time needed to compute the tasks in every machine is known [4], and that all the resources are free when the schedule starts. This is a well known problem, and it meets the typical necessities of many super-computing centers [4,14,37]. Finding the schedule that minimizes makespan is known to be NP-complete [18]. This points to the use of heuristic and metaheuristic algorithms.

Our algorithm, GraphCell, is a metaheuristic based on the Cellular Genetic algorithm (CGA) [1,30]. Before presenting our CGA, it is worth stating that the motivation behind GraphCell is not speedup, but finding better solutions (under an acceptable runtime). CGAs are known to find good solutions to many different optimization problems [1], but also originate from the adaptation of genetic algorithms to SIMD machines [8,15,30,33]; precisely the underlying architecture of the GPU. The CGA's good performance as metaheuristic and its suitability to the SIMD architecture make it a reasonable choice for the optimization problem considered.

It is a common assumption in the literature that the use of structured or decentralized populations in GAs allows a better exploration of the search space [1,2,6]. Distributed and cellular populations are the most common approaches of decentralized populations. In distributed GAs (DGAs) [2,6], the population is partitioned into a set of semi-isolated sub-populations (called islands) that are evolved by independent GAs. Periodically, the sub-populations exchange some information (typically, solutions) among them in a process called migration. This way, sub-populations are aware of the most promising locally explored regions in the other islands, and they can use that information in the search.

In the case of cellular GAs (CGAs) [1,30], the population is (usually) arranged into a two-dimensional toroidal grid, and only those solutions that are located next to each other in the lattice are allowed to interact during the application of the variation operators. The consequence is that solutions that are far from each other in the population are semi-isolated, and therefore, different regions of the population will hopefully explore distinct areas of the search space. The population is typically evolved as shown in Fig. 1. Every solution is sequentially updated by choosing the parents from its neighborhood. The variation operators are then applied to generate a new solution. Finally, the current solution is replaced by the new one if it is better. Therefore, using a cellular topology into the population restricts the number of solutions that can influence the evolution of every solution.

[*] Corresponding author.
E-mail address: bernabe.dorronsoro@uni.lu (B. Dorronsoro).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.02.018

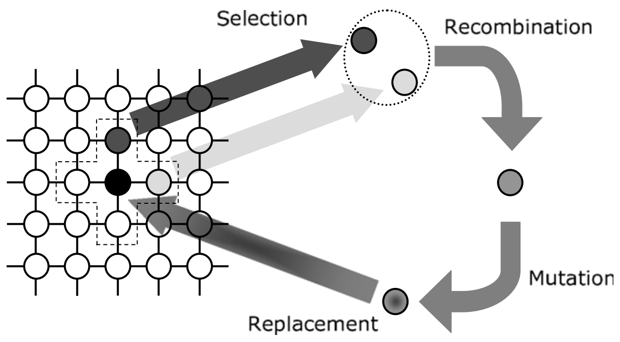


Fig. 1. How new solutions are generated in a cellular genetic algorithm.

The use of cellular populations in metaheuristics allows for a better exploration of the search space with respect to the equivalent one with other panmictic (i.e., non-decentralized) and decentralized populations. This is concluded in the literature for GAs [1,9], Particle Swarm Optimization (PSO) [21,22], and Differential Evolution (DE) [11], among others.

Finally, CGAs have successfully been applied to solve real world problems in domains like bioinformatics [39], telecommunications [41], logistics [10], image processing [31], or scheduling [17], to name a few.

It is however challenging to design such algorithms that find good solutions in a short execution time. In scheduling problems, we usually have a limited amount of time to find the best possible schedule of the tasks. Therefore, there is a need for algorithms that are able to find highly accurate solutions within this important time constraint.

The main contributions of this paper are (a) the design of a novel and efficient parallel Min–min heuristic [19] for the GPU and CPU, (b) two new highly parallel recombination operators for the GPU, and (c) GraphCell, a highly parallel CGA for GPU architectures. We analyze the performance of GraphCell with these two new operators, in isolation and interaction, based on a preliminary design [38].

The proposed new algorithm is highly competitive and efficient with respect to the state of the art for the studied problem instances, which range from smaller clusters of 16 processing units and 512 tasks to much larger ones of 2048 processing units and more than 65,000 tasks. Moreover, the GraphCell algorithm could be improved straightforwardly by adding a local search operator, an essential component in all the state-of-the-art techniques, that is not used in the current version. This is future work.

The new parallel Min–min algorithm shows high speedups with respect to the sequential Min–min heuristic and two equivalent parallel Min–min versions (using 4 and 8 threads) on the CPU. GraphCell is shown to be a valid alternative when accurate results are needed at the cost of longer computational times to schedule the tasks, outperforming Min–min for all problem instances and algorithm configurations, unlike its preliminary version MPS–CGA [38].

One interesting advantage of the implementation of the scheduler on the GPU, aside the excellent performance and low price of such devices, is that it does not require any computations to be done on the CPU. Therefore the load of the front-end in our system due to the scheduler is almost null, allowing it to process other requests.

The remainder of this paper is structured as follows. We give a brief overview of the main related works in Section 2. Then, we describe the problem at hands in Section 3. The new algorithms proposed and the results obtained are later described in Sections 4 and 5. Finally, we conclude the paper in Section 6.

2. Related work

The complexity of the independent task mapping problem confines candidate algorithms to heuristic and metaheuristic approaches, except for small problems. This section reviews past work on heuristic and metaheuristic algorithms for the GPU, applied to the problem considered. Then, we provide an overview of the main existing parallel CGAs on different architectures.

Solomon et al. [44] ported a metaheuristic algorithm called Particle Swarm Optimization (PSO) to the GPU, and applied it to the independent task mapping problem. They reported a speedup of 37 times over the sequential version. However, the solutions found by their algorithm are worse than those obtained with simpler heuristics, for problem instance sizes of 200 tasks $\times$ 40 machines and above.

Nesmachnow et al. [5] proposed GPU implementations of two scheduling heuristics, including Min–min. They report a maximum speedup of about 5 with respect to the sequential version of the heuristic. We obtain a much greater speedup, as presented in Section 5.2, although their GPU hardware seems comparable to ours. Their paper does not detail the parallel algorithm used, therefore we cannot explain the difference.

Van Luong et al. [46] investigated how multiobjective local search algorithms can be ported to the GPU for the flowshop scheduling problem. Their objective is speedup, and they reported an improvement of up to 16 times across the different local search algorithms.

We have mentioned in Section 1 that GraphCell is a CGA, and that CGAs are inherently parallel processes. Indeed, they initially appeared as an alternative design to panmictic GAs for SIMD machines, which is the underlying model of GPU. Some pioneer works in this line are those by Manderick and Spiessens [30,45], Mühlenbein [33,34], Gorges-Schleuter [15], and Collins [8].

With the popularity loss of massively parallel machines, some authors proposed different parallel implementations of cellular GAs, more appropriate for the distributed architectures that started to be available from the 90s.

In 1993, Maruyama et al. proposed in [32] a peculiar version of a parallel CGA for a cluster of machines in a local area network (LAN) in which single solutions are located in the processors, and after every generation solutions exchange information with only one randomly selected neighbor, as an attempt to reduce communications overhead.

After this first work on parallel CGAs for LAN architectures, there are a number of more recent papers proposing other designs that better fit the dynamics of the canonical sequential model. Nakashima et al. proposed in [35] a *combined CGA* where the population is divided into smaller square sub-populations, interacting through their borders. An image of this model can be seen on the right-hand side of Fig. 2, where the gray cells represent the solutions exchanged in the inter-processor communications. Folino et al. contributed in [13] with CAGE, a parallel cellular GP in which the population is divided into groups of columns (or rows) which constitute sub-populations (see the graph on the left in Fig. 2) to be run on different processors. This way they can reduce the number of messages with respect to the previous model, but messages will be bigger.

Luque et al. compared the performance of several parallel GAs in LAN environments [27]. Among them, both distributed and cellular GAs were the best performing ones, being the cellular algorithm slightly slower (from 3% to 10%) than the distributed one, but providing better solutions. Later, the authors analyzed different parallel CGA designs in [29], and proposed the use of asynchronous

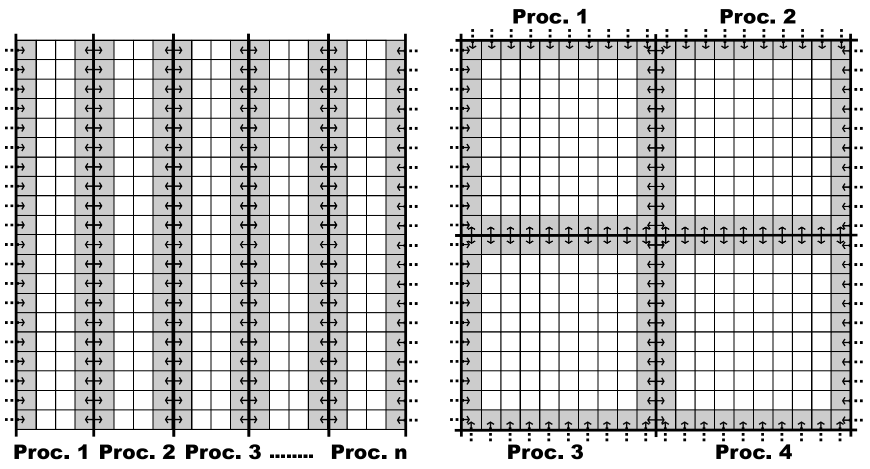


Fig. 2. CAGE (left) and the combined parallel model of CGA (right).

communications among processors in [28]. Dorronsoro et al. proposed PEGA in [10], a new parallel GA distributed in islands, with a CGA in every island, which can be executed either in local area network environments or in *computational grids*. PEGA was successfully applied to the largest existing instances of the VRP problem, contributing to the state of the art with some new solutions.

Pinel et al. proposed the first parallel implementations of CGAs for multi-core architectures, with applications to real-world problems as DNA sequencing [36] or scheduling [37]. In [39], the authors also analyzed several different strategies for the communications to deal with delays due to memory locks in parallel asynchronous CGAs on multi-core architectures.

Finally, there are a number of implementations of parallel CGAs in GPU architectures. The first works had to deal with complex data structures to map the algorithm data to texture rendering based on GPU, i.e., the information contained in the solutions must be allocated in the form of pixels in the GPU [23,25,51].

The appearance of tools like CUDA [42] or OpenCL [16] gave a major boost for the development of new parallel algorithms on GPU architectures. Among them, a few recent works are targeting cellular Evolutionary Algorithms (EAs). Soca et al. [43] proposed a framework for the implementation of cellular EAs on GPUs. Vidal and Alba also proposed a parallel version of CGA in a single GPU [48] and multiple ones [47]. Li et al. designed a fine-grained parallel immune algorithm [24]. Finally, Pinel et al. presented in [38] a parallel CGA for GPU architectures with a new recombination operator that is especially designed for good performance on the GPU. The current paper is an extension of that work.

To end this section, we summarize in Table 1 the main existing parallel cEAs proposed in the literature that have been discussed here.

3. Problem description

The problem addressed in this paper is how to assign independent tasks onto the different processors in a heterogeneous cluster, in order to minimize the makespan. Makespan is the completion time of the last machine (when the last machine finishes its tasks). Makespan is defined more formally later in this section. A machine is an independent computing unit, such as a single core in a multi-core processor.

This problem arises frequently in parameter sweep applications, such as Monte-Carlo simulations [7]. In these applications, many tasks with almost no interdependency are generated and submitted to a distributed system. In fact, more generally, the scenario in which the submission of independent tasks to a cluster is quite natural given that cluster users independently submit their tasks to the system and expect an efficient allocation of their tasks. We notice that efficiency means to allocate tasks as fast as possible and to optimize some criterion, such as makespan or flowtime. Makespan is among the most important optimization criteria of a distributed system; it is a measure of its productivity (throughput).

More precisely, assuming that the computing time needed to perform a task is known (assumption that is usually made in the literature [4,14,20]), we use the Expected Time to Compute (ETC) model by Braun et al. [4] to formalize an instance of the problem, as follows:

- A *number* of independent (user/application) *tasks* to be assigned.
- A *number* of heterogeneous *machine* candidates to participate in the planning. A machine is the general term for a computing unit, such as a core.
- The *workload* of each task (in millions of instructions).
- The *computing capacity* of each machine (in *mips*).
- Ready time indicating when machine m will have finished the previously assigned tasks. In this work, we consider, without loss of generality, that all the machines are available to process the assigned tasks ($\text{ready}_m = 0$).
- The expected time to compute (ETC) matrix (of size $nb\_\text{tasks} \times nb\_\text{machines}$) in which $\text{ETC}[t][m]$ is the expected execution time of task t on machine m.

We consider the task assignment as a single objective optimization problem, in which makespan is minimized. *Makespan*, the finishing time of latest task, is defined as:

$$\max\{\text{completion}[m] \mid m \in \text{Machines}\}, \tag{1}$$

where *completion* is the completion time of a machine. This time indicates when the machine will finalize the processing of the previous assigned tasks as well as of those already planned. Formally, for a machine m and a schedule S, the completion time of m is defined as follows:

$$\text{completion}[m] = \text{ready}_m + \sum_{t \in S^{-1}(m)} \text{ETC}[t][m]. \tag{2}$$

Table 1
Brief summary of the main existing parallel cEAs.

Authors	Reference		System	Comments
Manderick and Spiessens	[30]	(1989)	SIMD	Cellular GA with binary genes
Mühlenbein	[33]	(1989)	SIMD	Cellular GA for TSP
Gorges-Schleuter	[15]	(1989)	SIMD	CGA with Local search
Collins and Jefferson	[8]	(1991)	SIMD	Study of different selection schemes
Maruyama et al.	[32]	(1993)	LAN	Communications limited to one random neighbor
Nakashima et al.	[35]	(2002)	LAN	Population divided into square sub-populations
Folino et al.	[13]	(2003)	LAN	Population divided into rows/columns
Luque et al.	[27]	(2005)	LAN	Comparison versus other parallel GAs
Yu et al.	[51]	(2005)	GPU	Direct mapping of data structures to textures
Luo and Liu	[25]	(2006)	GPU	Direct mapping of data structures to textures
Dorronsoro et al.	[10]	(2007)	LAN/Grid	Distributed GA with CGAs in every island
Li et al.	[23]	(2007)	GPU	Direct mapping of data structures to textures
Luque et al.	[29]	(2009)	LAN	Design and comparison of several CGAs
Luque et al.	[28]	(2009)	LAN	Asynchronous parallel CGA
Li et al.	[24]	(2009)	GPU	Fine-grained parallel immune system
Pinel et al.	[37]	(2010)	Multi-core	Asynchronous parallel CGA
Pinel et al.	[39]	(2010)	Multi-core	Different communication policies study
Soca et al.	[43]	(2010)	GPU	Framework for cEAs using CUDA
Vidal and Alba	[48]	(2010)	GPU	Synchronous CGA implemented in CUDA
Vidal and Alba	[47]	(2010)	multiple GPUs	Synchronous CGA implemented in CUDA
Pinel et al.	[38]	(2010)	GPU	Synchronous parallel CGA implemented in CUDA

4. GraphCell, our proposed algorithm

This section describes GraphCell, the algorithm used in our experiments. GraphCell is a new parallel design of the CGA for the independent task assignment problem. A solution of the independent task assignment problem is the assignment of all the tasks to machines. The population is initialized with random solutions except for one, which is the result of the Min–min heuristic. This improves the search of good solutions. The main objective of GraphCell is to find accurate solutions, in a reasonable time. Given the prohibitive runtime of the Min–min heuristic on large problem instances, we require a parallel version of the Min–min heuristic on the GPU.

Section 4.1 presents the Min–min heuristic and our parallel version for the GPU. Section 4.2 describes the new parallel CGA for GPU we propose.

4.1. Min–min heuristic for the scheduling problem

The Min–min heuristic is a simple deterministic algorithm initially proposed by Ibarra and Kim in 1977 [19] for the scheduling problem of independent tasks. Due to its accuracy and simplicity, the algorithm has been used as a reference in many research papers since then [4,26,40] or as a component for the design of more efficient algorithms [12,49,50].

The Min–min algorithm iteratively proceeds in three steps. First, it finds the best machine assignment for each unassigned task (the first "min"). Here, best means minimal completion time. Second, it chooses among all the previous possible assignments, the one with the minimum completion time (the second "min"). Finally, it assigns that task to the corresponding machine. The process continues until all tasks have been assigned.

Our proposed GPU implementation for the Min–min algorithm is presented in Algorithm 1. The `f <<< n >>>()` notation reflects the CUDA macros: it indicates that kernel `f` is launched across `n` threads. The first step is the launch of the `min_ct` kernel. For each task, a thread finds the best machine for a given task, by selecting the machine with the minimum estimated completion time. This kernel is launched with `Tasks` threads, because the selection of the best machine can be conducted in parallel. Threads of previously assigned tasks are also run, but immediately return from the kernel. Then, the results are copied, from device memory to a temporary area on device memory, for the parallel reduction. The parallel reduction presented here (lines 5–9) is a simplified

Algorithm 1 Pseudo-code for Min–min heuristic on the GPU

```
1:  for all Tasks of one solution do
2:     $min\_ct <<< Tasks >>> (results)$ // Step 1
3:     cudaMemcpy (results, temp, cudaMemcpyDeviceToDevice)
4:     // Step 2: parallel reduction
5:     $n \leftarrow 2$
6:     while $Tasks/n \geq 1$ do
7:        $min\_task <<< Tasks/n >>> (temp)$
8:        $n \leftarrow n \times 2$
9:     end while
10:    $assign <<< 1 >>> (temp, solution)$ // Step 3
11: end for
```

version of the code actually used to identify the best (minimal) task/machine assignment. Finally, one thread runs the `assign` kernel to update the solution with the best assignment. When all tasks have been assigned, the solution found can either be copied to the host memory or kept into the device memory, depending if the algorithm is run alone or as part of GraphCell. In the latter case, the solution found by the heuristic is directly used by the parallel CGA on the GPU, which is presented next.

4.2. Parallel synchronous CGA

GraphCell is a highly parallel synchronous cellular genetic algorithm for GPU architectures. It uses two new recombination operators. In order to study the effect of these operators, in isolation and interaction, they are combined into a single operator. The combination is called Uniform Proportional Recombination (UPR). The two recombination operators of UPR are specifically designed for algorithms implementing cellular topologies in massively parallel architectures like GPUs. Indeed, we depart from the usual design where one solution is evolved by a single GPU thread. Instead, both recombination operators are run with one thread per task of a solution. Also, each solution of the population is recombined in parallel. This leads to a high number of threads, especially when larger problem instances are experimented. The details are presented in Algorithm 2, and a description is provided below.

Fig. 3 shows how the recombination operators update each task of a solution. The arrays shown represent the solutions, each cell of the array corresponds to a task. The number in a cell of the array is the machine to which this task is assigned (denoted by

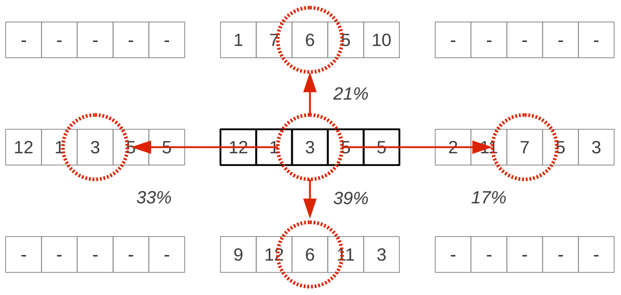


Fig. 3. Common design of the two parallel recombination operators.

the array's index). The circled task of the center solution shows the task being updated. Nine solutions are shown. The solutions directly above, below, to right of, to the left of, the center solution define the neighboring solutions. The other solutions are ignored by both recombination operators. The new machine assignment for the circled task is computed by a dedicated thread (if Tasks is the number of tasks of a solution, Tasks threads are run to compute the new solution). The two proposed recombination operators follow the same design: the offspring solution is generated by assigning to each task, the machine of one of the neighboring solutions, according to a proportionate selection mechanism. The operators only differ in the criterion used for this selection:

- Fitness (UPR_f): we choose the assignment for each task of one neighboring solution with a proportionate selection mechanism based on the fitness of the solutions (i.e., the probability for choosing one neighboring solution is given by its fitness value over the sum of the fitness of all the neighbors). Therefore, parent i will be chosen with probability:

$$\text{PF}_{\text{Solution}_i} = \frac{\text{Fitness}(\text{Solution}_i)}{\sum_{j=0}^{\text{No. Solution}} \text{Fitness}(\text{Solution}_j)}.$$

- Completion time (UPR_{ct}): we choose the assignment for each task of one neighboring solution with a proportionate selection mechanism based on the estimated time to complete on the machine to which the considered task is assigned in the neighboring solution. In this case, the probability of choosing the assignment of neighboring solution i is:

$$\text{PCT}_{\text{Solution}_i} = \frac{\text{ETC}_{i,j}}{\sum_{k=0}^{\text{No. Solution}} \text{ETC}_{k,j}},$$

  where $\text{ETC}_{i,j}$ is the execution time of task j to the machine to which it is assigned in neighboring solution i. This value is given by the ETC matrix of the considered instance.

We study different combinations of these operators by defining the probability (P_{sel}) to apply the fitness based operator. The probability to apply the completion time operator is $1 - P_{\text{sel}}$. This decision is made for each task, and not for the entire solution.

The percentages shown in Fig. 3 highlight this selection mechanism. From this description, we notice that the total number of threads used for the recombination UPR is: population size $\times$ solution size (which is the total number of tasks). This generates a high number of lightweight threads (more than 10^6), which is well suited to the GPU. Algorithm 2 presents the pseudo-code for GraphCell. GraphCell, the Min–min heuristic and the CGA, is executed only on the GPU.

GraphCell initializes the population of solutions randomly (with a uniform distribution), except for one solution which is the result of the Min–min heuristic, as presented in Section 4.1.

Algorithm 2 Pseudo-code of GraphCell

```
1:  // Population initialization:
2:  // First, initialize one solution with Min–min:
3:  for all Tasks of one solution do
4:      $min\_ct <<< Tasks >>> (results)$
5:      cudaMemcpy (results, temp, cudaMemcpyDeviceToDevice)
6:      $n \leftarrow 2$
7:      while $Tasks/n \geq 1$ do
8:          $min\_task <<< Tasks/n >>> (temp)$
9:          $n \leftarrow n \times 2$
10:     end while
11:     $assign <<< 1 >>> (temp, solution)$
12: end for
13: // The rest of population is initialized randomly.
14: // The CGA:
15: while not $stop\_condition()$ do
16:     $neighborhood\_prob <<< Pop >>> ()$
17:     $upr <<< Pop \times Tasks >>> ()$
18:     $mutate <<< Pop >>> ()$
19:     $fitness <<< Pop >>> ()$
20:     $replace <<< Pop >>> ()$
21: end while
```

UPR is implemented in the two GPU kernels `neighborhood_prob`, and `upr`. Kernel `neighborhood_prob` computes the probability for each solution in the neighborhood to be chosen under fitness proportionality. This kernel is run by one thread per solution, because the fitness is defined per solution. The kernel `upr` randomly selects (with a uniform distribution) the recombination operator (UPR_f or UPR_{ct}) to apply for a task, according to probability P_{sel}. Then, the kernel randomly chooses a task assignment among the same tasks of neighborhood solutions, applying the proportionate selection mechanism. It computes the different probabilities for the estimated completion time proportionality, if needed, on demand because this is task dependent. Kernel `upr` is run by one thread per task per solution, because the assignment decision for a task is independent of all the other tasks. The other kernels, `mutate`, `fitness`, and `replace` are launched with one thread per solution. Kernel `mutate` changes the assignment of a randomly chosen task, to a randomly chosen machine. Kernel `fitness` computes the makespan for the solution. Kernel `replace` replaces the old solution with the new computed solution if it is not worse (in terms of makespan). These last three kernels are standard operators in genetic algorithms.

The stopping condition can either be a maximum number of evaluations, or time (wall-clock).

5. Experimentation

In this section, we study the performance of the Min–min and GraphCell algorithms. First, we describe in Section 5.1 the problem instances generated for the simulations. Then, we evaluate in Section 5.2 the GPU version of Min–min heuristic proposed. We present in Section 5.3 the GraphCell configuration and its performance (in terms of solution quality) on problem instances ranging from 512 tasks over 16 machines, to 65,536 tasks over 2048 machines.

The computer used in the experiment is a Dell Precision T5400, which includes two Intel Xeon E5440 processors (dual processor, of 4 cores each), clocked at 2.83 GHz, with 16 GiB of main memory. The computer runs the GNU/Linux operating system Ubuntu Server (64-bit kernel, version 2.6.35-27). The GPU installed on this computer is a Nvidia Tesla C2050, with CUDA driver version 3.20 (capability 2.0). This GPU holds 14 multi-processors (of 32

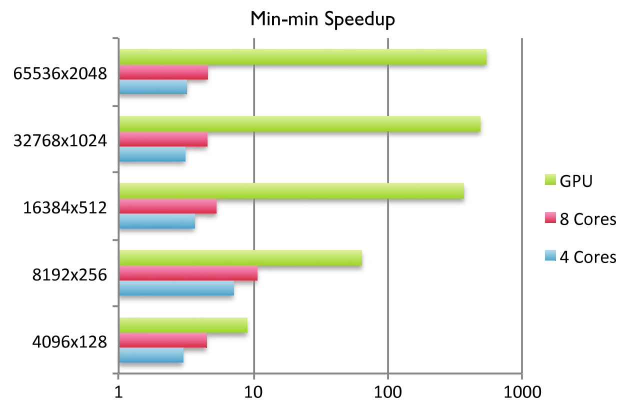


Fig. 4. Speedup results of the GPU versus the equivalent sequential and parallel CPU Min–min (logarithmic scale).

cores each), clocked at 1.15 GHz, and with a global memory of 2 GiB. All programs are written in C, except for the GPU kernels which are written in CUDA C. The operating system's Pthread library is used for the multi-threaded versions of the parallel CPU versions.

5.1. *Problem instances*

As previously mentioned, Section 3, we assume that the estimated computing time needed to perform each task on each machine is known. The problem can therefore be represented by the expected time to compute (ETC) model, as suggested by Braun et al. [4].

We study six different instance sizes, specifically 512×16, 4096×128, 8192×256, $16{,}384 \times 512$, $32{,}768 \times 1024$, and $65{,}536 \times 2048$, where the first figure is the number of tasks and the second the number of machines the tasks must be assigned to. A machine is an independent computing unit, such as a core in multi-core architectures.

The chosen problem instances are generated with high task and machine heterogeneity,[1] which we consider realistic. This reflects different tasks and different processor types. Large problem instances should reflect different processor types, as a cluster is often the result of several machine acquisitions.

The instances were randomly generated (we created them as described in [3], using R), therefore 20 different instances were considered for every problem size in our experiments.

5.2. *Performance evaluation of parallel Min–min*

We evaluate in this section the performance of the parallel Min–min design presented in Section 4.1. Because the Min–min heuristic is a deterministic algorithm, and the parallel Min–min performs exactly the same search as the sequential Min–min, we only compare them by means of execution time. We implemented three different parallel versions of the algorithm: two multi-threaded CPU implementations, using 4 and 8 cores, and the GPU implementation.

We show in Fig. 4 the speedup results of the different parallel Min–min implementations. The speedup is measured as the time the sequential Min–min takes on the CPU over the time of the corresponding algorithm. Notice that the *x* axis is represented in logarithmic scale. As it can be seen, the two parallel Min–min algorithms scale well with the problem size, providing similar speedup values for all of them. However, the algorithm does not scale so well with the number of cores. This is probably due to memory contention. As it can be seen, the parallel Min–min using 8 cores is always faster than the equivalent version with 4 cores, but the average speedup increases from 4.024 to 5.918 when doubling the number of cores from 4 to 8. It is worth emphasizing that the parallel Min–min on 4 cores is achieving linear speedups in average for the considered problem sizes. Additionally, we notice that the algorithm performs super-linear speedups for the 8192 tasks instance: 7.11 for 4 cores and 10.67 for 8. We suspect this is due to higher cache hit ratios thanks to memory sharing.

We now turn to the results of the GPU version. We can see that its performance is clearly higher with respect to the CPU versions, with speedups ranging from 9 for the smallest instance to 538 for the biggest one (the sequential Min–min algorithm takes more than 3 days to find a solution for this instance). Additionally, the algorithm scales well with the problem size, since the bigger the problem, the higher the speedup obtained. Therefore, the performance of the parallel GPU version with respect to the parallel CPU ones is better when the problem size increases.

At this point, we should mention that the GPU version is slightly different than the parallel CPU ones. They differ in the way the second step of the Min–min algorithm is implemented, where it searches for the task that is earlier accomplished among those chosen in the first step. In the CPU version, tasks are iteratively traversed to choose the earliest accomplished one, while in the GPU algorithm this is done with parallel reduction, benefiting from the massively parallelization provided by the GPU, which requires $\log_2$ Tasks iterations (cf., Section 4.1).

The GPU program only uses the global device memory, therefore additional speedup may be achievable using the faster GPU memories, especially for the read only instance matrices. Also, the default parameter settings for the cache size were used.

5.3. *Performance evaluation of GraphCell*

In this section, we investigate the performance of our new highly parallel synchronous cellular genetic algorithm, GraphCell. First, we detail in Section 5.3.1 the configuration used in the algorithm. Then, Section 5.3.2 analyzes the performance results of GraphCell.

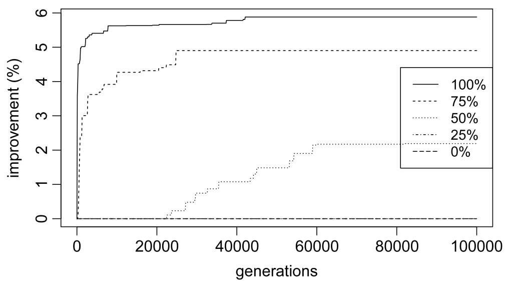
5.3.1. *Configuration of algorithms*

Table 2 presents the configuration of the algorithms. The population was set to 8×8 solutions after some experimentation with two other larger ones (16×16 and 32×32). The population is randomly initialized, except for one solution that is generated using the Min–min heuristic. The neighborhood used is von Neumann, also called L5. The recombination operators are combined into the new Uniform Proportional Recombination (UPR). The solutions are encoded in an integer array of length the number of tasks, where the content of a cell is the machine to which the task (denoted by the array index) is assigned. The mutation operator consists of assigning a random machine to a task with probability one over the number of tasks. Finally, new solutions replace previous solutions in-place, if their fitness is less or equal than the previous solutions. The execution stops after 100,000 generations (each solution is evolved 100,000 times).

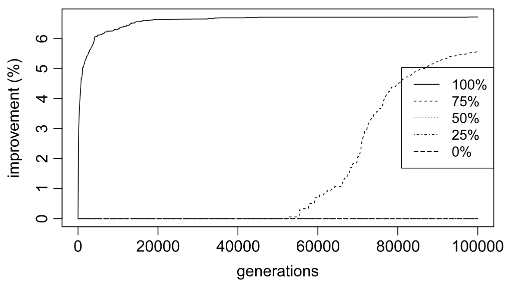
5.3.2. *Results*

In this section, we analyze the quality of the solutions found by GraphCell, across different problem sizes. The effect of each of the two recombination operators is explored by varying the UPR parameter P_{sel}. When $P_{sel} = 1$, then the fitness proportionate selection is chosen (i.e., UPR_f). When $P_{sel} = 0$, then the estimated completion time proportionate selection is chosen (i.e., UPR_{ct}): other values reflect the interaction of the two operators.

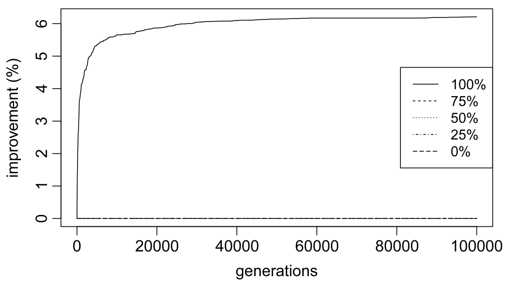
[1] Instances available in: http://par-cga-sched.gforge.uni.lu/instances.



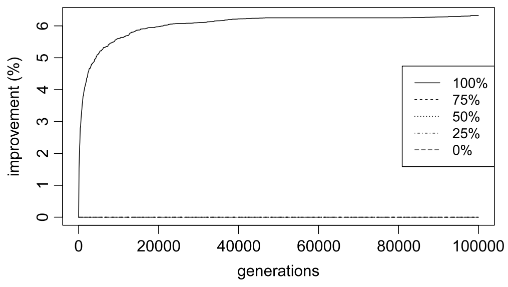
(a) 512 tasks × 16 machines.



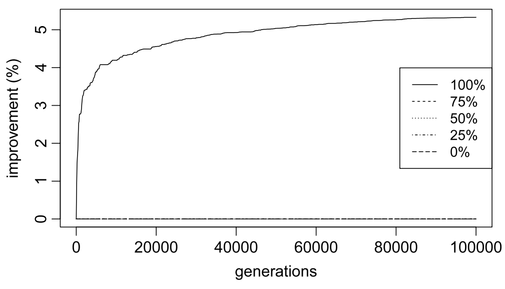
(b) 4096 tasks × 128 machines.



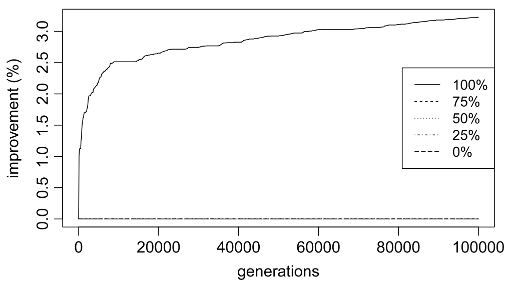
(c) 8192 tasks × 256 machines.



(d) 16,384 tasks × 512 machines.



(e) 32,768 tasks × 1024 machines.



(f) 65,536 tasks × 2048 machines.

Fig. 5. Improvement of best solution, compared to Min–min solution.

Table 2
Configuration for GraphCell.

Population size	8 × 8 solutions
Population initialization	1 solution with Min–min and the rest random
Neighborhood	von Neumann
Recombination	Uniform Proportional Recombination (UPR)
Recombination probability	$p_r = 1.0$
Mutation	Random
Mutation probability	$p_m = 1.0/\text{numberOfTasks}$
Replacement	Replace if Better or Equal
Termination condition	100,000 generations

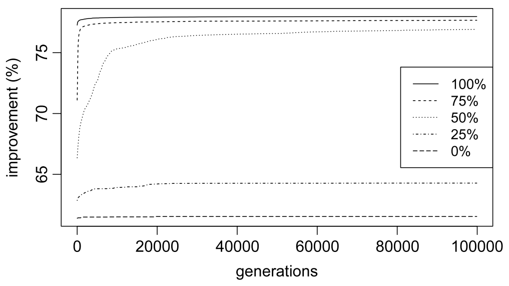
The results shown were obtained after 20 independent runs of the algorithm, solving a different problem instance every time.

Fig. 5 shows the evolution of the improvement of the best solution in the population (averaged over the 20 instances), compared to the solution generated with the Min–min heuristic, for the five different configurations of UPR. These configurations are the different values of the P_{sel} probability. These values are $P_{\text{sel}} = 1$, 0.75, 0.5, 0.25, and 0. Therefore, these plots show how the performance of the CGA part of GraphCell improves upon Min–min with the five considered UPR configurations. We can observe that increasing the weight of the fitness proportionate operator improves the performance of the algorithm. Indeed, from the first generations the algorithms using $P_{\text{sel}} = 1$ (this is UPR$_f$) and 0.75 are able to improve Min–min solution by 3%, while the original algorithm with $P_{\text{sel}} = 0.5$ is not able to achieve this improvement in the 100,000 generations allowed.

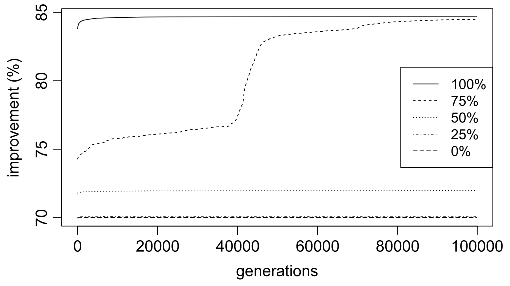
Therefore, GraphCell improves very quickly the initial Min–min solution, and the improvement continues until the end of the run, especially for the biggest instances.

It is well known that increasing the neighborhood size leads to faster convergence speeds in cellular evolutionary algorithms [1]. Therefore we tried to accelerate the convergence speed of the algorithm by employing a larger neighborhood, namely C13, composed by the solution itself plus the 12 nearest ones in Manhattan distance. We ran the GraphCell algorithm for 50,000 generations with the two neighborhood structures on the 20 instances of each of the problem sizes, but did not notice any statistically significant differences (according to the unpaired Wilcoxon signed rank test) on the quality of solutions found (the p-values obtained were 0.4291, 0.718, 0.698, 0.57, and 0.5291 for the instances from smaller to larger, respectively). However, the runtime of the program increased with the neighborhood size.

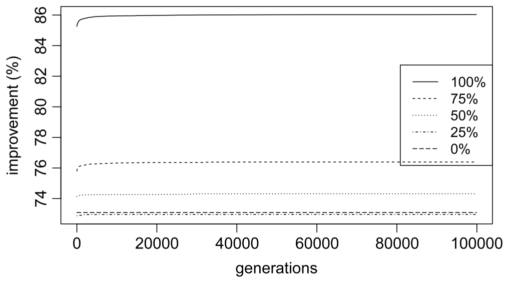
We now plot in Fig. 6 the evolution of the average fitness over the population for the GraphCell algorithm with the five different UPR configurations. Since the population is initialized with random solutions, except one with the Min–min heuristic, the initial average fitness is poor. We see that the results are similar to the ones discussed in the previous experiment. The value of P_{sel} has a significant impact on the improvement of the population during the run, and the larger the number of tasks to schedule, the more



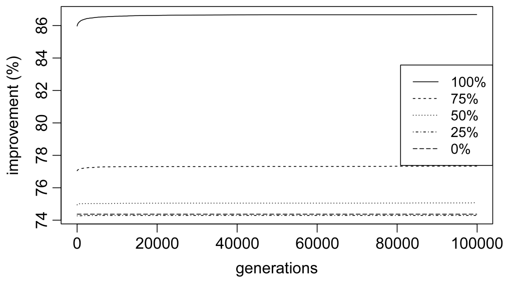
(a) 512 tasks × 16 machines.



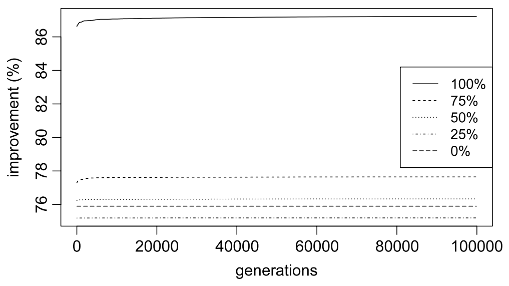
(b) 4096 tasks × 128 machines.



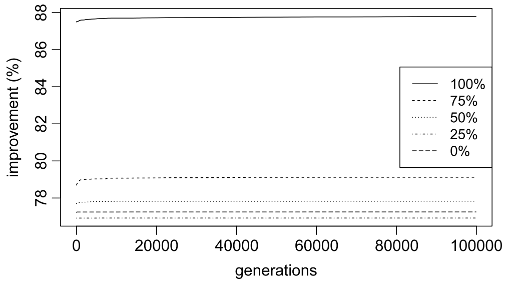
(c) 8192 tasks × 256 machines.



(d) 16,384 tasks × 512 machines.



(e) 32,768 tasks × 1024 machines.



(f) 65,536 tasks × 2048 machines.

Fig. 6. Improvement of fitness average across population, compared to fitness average across initial population.

important the difference with $P_{\text{sel}} = 1$ is. With respect to the other configurations, only for the smallest instances two of them are able to perform significant improvements on the average quality of the solutions in the initial population: $P_{\text{sel}} = 0.75$ for 512 and 4096 tasks, and $P_{\text{sel}} = 0.5$ for 512 tasks.

6. Conclusion and future work

In this paper, we presented the parallel Min–min heuristic for the GPU, and GraphCell, an extension of the preliminary work MPS–CGA, introduced in [38]. In that paper, we presented a new GPU implementation of a CGA with a new recombination operator for the scheduling problem. It was demonstrated to be faster and more accurate than another highly specialized state-of-the-art parallel CGA on the CPU, for most classes of small problem sizes.

In this work, we extended our preliminary algorithm with a new design for a parallel implementation of Min–min algorithm both on a multi-core CPU and a GPU. The design proposed in this work is a fast and accurate state-of-the-art heuristic for scheduling of independent tasks. The new GPU parallel Min–min provides high speedup values and better scalability with respect to the sequential and multi-threaded versions.

Additionally, we explored the two recombination operators designed for the GPU, and their interaction. In an extensive study, five different configurations of these operators were analyzed, and we discovered that the best performing operator was the one relying on the fitness value of the solutions. Moreover, two different neighborhoods were studied, we did not find a significant influence on the quality of the solutions. GraphCell uses this new recombination operator and the GPU parallel Min–min algorithm in the population initialization.

As future work, we are investigating the use of efficient local search heuristics for this problem on the GPU. We believe that adding a local search operator could help to improve our results, as it is the case of most algorithms in the literature for this problem. Also, the other GraphCell operators (i.e., mutation) could be modified to exploit greater parallelism.

Acknowledgment

The work was supported by the National Research Fund, Luxembourg, Project Green-IT C09/IS/05.

References

[1] E. Alba, B. Dorronsoro, Cellular Genetic Algorithms, in: Operations Research/Compuer Science Interfaces, Springer-Verlag, Heidelberg, 2008.
[2] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Transactions on Evolutionary Computation 6 (5) (2002) 443–462.
[3] S. Ali, H.J. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Representing task and machine heterogeneities for heterogeneous, Journal of Science and Engineering 3 (2000) 195–207. Special 50th Anniversary Issue.

[4] T.D. Braun, H.J. Siegel, N. Beck, L.L. Bölöni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen, R.F. Freud, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810–837.

[5] S. Nesmachnow, M. Canabé, GPU implementations of scheduling heuristics for heterogeneous computing environments, in: Proceedings of the XVII Congreso Argentino de Ciencias de la Computación, 2011, pp. 1563–1570. URL: http://www.fing.edu.uy/inco/cursos/hpc/material/clases/gpu_hcsp_heuristics.pdf.

[6] E. Cantú-Paz, Efficient and Accurate Parallel Genetic Algorithms, second ed., in: Book Series on Genetic Algorithms and Evolutionary Computation, vol. 1, Kluwer Academic Publishers, 2000.

[7] H. Casanova, A. Legrand, D. Zagorodnov, F. Berman, Heuristics for scheduling parameter sweep applications in grid environments, in: Heterogeneous Computing Workshop, 2000, pp. 349–363.

[8] R. Collins, D. Jefferson, Selection in massively parallel genetic algorithms, in: R. Belew, L. Booker (Eds.), Proc. of the Fourth International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann, San Diego, CA, USA, 1991, pp. 249–256.

[9] G. Danoy, B. Dorronsoro, P. Bouvry, Overcoming partitioning in large ad hoc networks using genetic algorithms, in: Proc. of the Genetic and Evolutionary Computation COnference (GECCO), ACM, 2009, pp. 1347–1354.

[10] B. Dorronsoro, D. Arias, F. Luna, A. Nebro, E. Alba, A grid-based hybrid cellular genetic algorithm for very large scale instances of the CVRP, in: W.W. Smari (Ed.), High Performance Computing & Simulation Conference (HPCS), 2007, pp. 759–765.

[11] B. Dorronsoro, P. Bouvry, Improving classical and decentralized differential evolution with new mutation operator and population topologies, IEEE Transactions on Evolutionary Computation 15 (1) (2010) 67–98.

[12] B. Dorronsoro, P. Bouvry, J.A. Cañero, A.A. Maciejewski, H.J. Siegel, Multi-objective robust static mapping of independent tasks on grids, in: Proceedings of the IEEE Congress on Evolutionary Computation (CEC), part of World Conference in Computational Intelligence (WCCI), 2010, pp. 3389–3396.

[13] G. Folino, C. Pizzuti, G. Spezzano, A scalable cellular implementation of parallel genetic programming, IEEE Transactions on Evolutionary Computation 7 (1) (2003) 37–53.

[14] A. Ghafoor, J. Yang, Distributed heterogeneous supercomputing management system, IEEE Computer 26 (6) (1993) 78–86.

[15] M. Gorges-Schleuter, ASPARAGOS—an asynchronous parallel genetic optimization strategy, in: J. Schaffer (Ed.), Proc. of the Third International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann, 1989, pp. 422–428.

[16] Khronos Group, Open computing language, http://www.khronos.org/opencl/.

[17] M. Guzek, J. Pecero, B. Dorronsoro, P. Bouvry, A cellular genetic algorithm for scheduling applications and energy-aware communication optimization, in: Workshop on Optimization Issues in Energy Efficient Distributed Systems (OPTIM), part of the International Conference on High Performance Computing & Simulation (HPCS), 2010, pp. 241–248.

[18] E. Horowitz, S. Sahni, Exact and approximate algorithms for scheduling nonidentical processors, Journal of the ACM 23 (1976) 317–327.

[19] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, Journal of the ACM 24 (2) (1977) 280–289.

[20] M. Kafil, I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurrency 6 (3) (1998) 42–51.

[21] J. Kennedy, R. Mendes, Population structure and particle swarm performance, in: IEEE International Conference on Evolutionary Computation (CEC), 2002, pp. 1671–1676.

[22] J. Kennedy, R. Mendes, Neighborhood topologies in fully informed and best-of-neighborhood particle swarms, IEEE Transactions on Systems, Man, and Cybernetics—Part C: Applications and Reviews 36 (4) (2006) 515–519.

[23] J.-M. Li, X.-J. Wang, R.-S. He, Z.-X. Chi, An efficient fine-grained parallel genetic algorithm based on GPU-accelerated, in: IFIP International Conference on Network and Parallel Computing, IEEE, 2007, pp. 855–862.

[24] J. Li, L. Zhang, L. Liu, A parallel immune algorithm based on fine-grained model with GPU-acceleration, in: Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control, IEEE Press, 2009, pp. 683–686.

[25] Z. Luo, H. Liu, Cellular genetic algorithms and local search for 3-SAT problem on graphic hardware, in: IEEE Congress on Evolutionary Computation, 2006, pp. 10345–10349.

[26] P. Luo, K. Lu, Z. Shi, A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing 67 (2007) 695–714.

[27] G. Luque, E. Alba, B. Dorronsoro, Parallel Metaheuristics: A New Class of Algorithms, Wiley, 2005, pp. 107–125. Parallel genetic algorithms (Chapter).

[28] G. Luque, E. Alba, B. Dorronsoro, An asynchronous parallel implementation of a cellular genetic algorithm for combinatorial optimization, in: Proceedings of the International Genetic and Evolutionary Computation Conference (GECCO), ACM, 2009, pp. 1395–1402.

[29] G. Luque, E. Alba, B. Dorronsoro, Optimization Techniques for Solving Complex Problems, Wiley, 2009, pp. 49–62. Analyzing parallel cellular genetic algorithms (Chapter).

[30] B. Manderick, P. Spiessens, Fine-grained parallel genetic algorithm, in: J. Schaffer (Ed.), Third International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann, 1989, pp. 428–433.

[31] T. Mantere, Object and pose recognition with cellular genetic algorithms, in: Intelligent Robots and Computer Vision XXV: Algorithms, Techniques, and Active Vision, vol. 6764, Optics East 2007, SPIE, 2007, 67640N-1-10.

[32] T. Maruyama, T. Hirose, A. Konagaya, A fine-grained parallel genetic algorithm for distributed parallel systems, in: Proc. of the Fifth International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann, San Francisco, CA, USA, 1993, pp. 184–190.

[33] H. Mühlenbein, Parallel genetic algorithms, population genetic and combinatorial optimization, in: Proc. of the Third International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann, 1989, pp. 416–421.

[34] H. Mühlenbein, M. Gorges-Schleuter, O. Krämer, Evolution algorithms in combinatorial optimization, Parallel Computing 7 (1988) 65–88.

[35] T. Nakashima, T. Ariyama, H. Ishibuchi, Combining multiple cellular genetic algorithms for efficient search, in: Proc. of the Asia-Pacific Conference on Simulated Evolution and Learning (SEAL), 2002, pp. 712–716.

[36] F. Pinel, B. Dorronsoro, P. Bouvry, A new parallel asynchronous cellular genetic algorithm for de novo genomic sequencing, in: Proceedings of the 2009 IEEE International Conference of Soft Computing and Pattern Recognition, IEEE Press, 2009, pp. 178–183.

[37] F. Pinel, B. Dorronsoro, P. Bouvry, A new parallel asynchronous cellular genetic algorithm for scheduling in grids, in: Nature Inspired Distributed Computing (NIDISC) sessions of the International Parallel and Distributed Processing Simposium (IPDPS) 2010 Workshop, IEEE Press, 2010, pp. 1–8. http://dx.doi.org/10.1109/IPDPSW.2010.5470698.

[38] F. Pinel, B. Dorronsoro, P. Bouvry, A new cellular genetic algorithm to solve the scheduling problem designed for the GPU, in: Metaheuristics Conference (META), 2010, eProceedings.

[39] F. Pinel, B. Dorronsoro, P. Bouvry, On the parallelization of asynchronous cellular genetic algorithms for multi-core architectures, in: International Conference of the Association of Latin-Iberoamerican Operational Research Societies, Institute for Operations Research and the Management Sciences (ALIO/INFORMS) Joint International Meeting, 2010, Abstract communication.

[40] G. Ritchie, J. Levine, A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments, in: 23rd Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2004), 2004.

[41] P. Ruiz, B. Dorronsoro, G. Valentini, F. Pinel, P. Bouvry, Optimisation of the enhanced distance based broadcasting protocol for MANETs, Journal of Supercomputing (2011). http://dx.doi.org/10.1007/s11227-011-0564-x.

[42] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, NVIDIA Corporation, 2011.

[43] N. Soca, J.L. Blengio, M. Pedemonte, P. Ezzatti, PUGACE, a cellular evolutionary algorithm framework on GPUs, in: IEEE Congress on Evolutionary Computation, 2010, eProceedings.

[44] S. Solomon, P. Thulasiraman, R. Thulasiram, Collaborative multi-swarm PSO for task matching using graphics processing units, in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO'11, ACM, New York, NY, USA, 2011, pp. 1563–1570.

[45] P. Spiessens, B. Manderick, A massively parallel genetic algorithm: implementation and first analysis, in: R. Belew, L. Booker (Eds.), Proc. of the Fourth International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann, 1991, pp. 279–286.

[46] T. Van Luong, N. Melab, E.-G. Talbi, GPU-based approaches for multiobjective local search algorithms. A case study: the flowshop scheduling problem, in: P. Merz, J.-K. Hao (Eds.), Evolutionary Computation in Combinatorial Optimization, in: Lecture Notes in Computer Science, vol. 6622, Springer, Berlin, Heidelberg, 2011, pp. 155–166.

[47] P. Vidal, E. Alba, A multi-GPU implementation of a cellular genetic algorithm, in: IEEE Congress on Evolutionary Computation, 2010, eProceedings.

[48] P. Vidal, E. Alba, Nature Inspired Cooperative Strategies for Optimization (NICSO), in: Studies in Computational Intelligence (SCI), vol. 284, Springer, 2010, pp. 223–232. Cellular genetic algorithm on graphic processing units (Chapter).

[49] F. Xhafa, E. Alba, B. Dorronsoro, B. Duran, Efficient batch job scheduling in grids using cellular memetic algorithms, Journal of Mathematical Modelling and Algorithms 7 (2) (2008) 217–236.

[50] F. Xhafa, J. Carretero, E. Alba, B. Dorronsoro, Design and evaluation of tabu search method for job scheduling in distributed environments, in: Nature Inspired Distributed Computing (NIDISC) sessions of the International Parallel and Distributed Processing Simposium (IPDPS) 2008 Workshop, IEEE Press, 2008, pp. 2319–2326.

[51] Q. Yu, C. Chen, Z. Pan, Advances in Natural Computation, in: Lecture Notes in Computer Science (LNCS), vol. 3612, Springer, 2005, pp. 1051–1059. Parallel genetic algorithms on programmable graphics hardware (chapter).

Frédéric Pinel is a Ph.D. candidate at the University of Luxembourg. He is currently working on evolutionary algorithms applied to parallel computing and energy-efficiency. He holds Master's degrees from FUNDP, Belgium (2005) and ESIGELEC, France (1991).

Bernabé Dorronsoro received the degree in engineering (2002) and the Ph.D. in Computer Science (2007) from the University of Málaga (Spain), and he is currently working as research associate at the University of Luxembourg. His main research interests include Grid computing, ad hoc networks, the design of new efficient metaheuristics, and their application for solving complex real-world problems in the domains of logistics, telecommunications, bioinformatics, combinatorial, multiobjective, and global optimization. Among his main successful publications, he has several articles in impact journals and one book. Dr. Dorronsoro has been member of the organizing committees of several conferences and workshops, and he usually serves as reviewer for leading impact journals and conferences.

Pascal Bouvry earned his Ph.D. degree ('94) in computer science at the University of Grenoble, France. He is now Professor at the Faculty of Sciences, Technology and Communication of the University of Luxembourg and heading the Computer Science and Communication research unit (http://csc.uni.lu). Pascal Bouvry is specialized in parallel and evolutionary computing. His current interest concerns the application of nature-inspired computing for solving reliability, security, and energy-efficiency problems in clouds, grids and ad hoc networks.