A parallel online GPU scheduler for large heterogeneous computing systems

Santiago Iturriaga[1], Sergio Nesmachnow[1], Francisco Luna[2], and Enrique Alba[2]

[1] Universidad de la República, Uruguay
{siturria,sergion}@fing.edu.uy
[2] Universidad de Málaga, Spain
{flv,eat}@lcc.uma.es

Abstract. This work presents a parallel implementation on GPU for a stochastic local search method to efficiently solve the task scheduling problem in heterogeneous computing environments. The research community has been searching for accurate schedulers for heterogeneous computing systems, able to run in reduced times. The parallel stochastic search proposed in this work is based on simple operators in order to keep the computational complexity as low as possible, thus allowing large scheduling instances to be efficiently tackled. The experimental analysis demonstrates that the parallel stochastic local search method on GPU is able to compute accurate suboptimal schedules in significantly shorter execution times than state-of-the-art schedulers.

Keywords: GPU computing, heterogeneous computing, scheduling.

1 Introduction

Scheduling tasks in current heterogeneous computing (HC) systems challenges researchers to the problem of assigning dozens of thousands of tasks in very short times that should be limited to a few seconds. Indeed, HC systems are becoming larger and larger during the last fifteen years, mainly due to the fast increase of computing power and the rapid development of high-speed networking protocols, but also to the demand of the scientific community that has to address increasingly large problems that require an enormous computing power [7]. Assigning and mapping tasks becomes therefore a critical issue since finding an accurate schedule significantly impacts in both the resource utilization costs and the quality of service (QoS) perceived by the user.

Scheduling problems have been widely studied in operational research [4,12], but most of the classical approaches have faced the task scheduling in homogeneous environments. The ultimate goal of a scheduling problem is to provide an assignment of tasks to resources so that some efficiency criteria is satisfied, usually related to the total execution time for a bunch of tasks (makespan), but frequently also considering other metrics such as the resource utilization and/or the QoS.

The *Heterogeneous Computing Scheduling Problem* (HCSP), in which the resources are different among them, has become important due to the popularization of distributed computing and the growing use of heterogeneous clusters [5,8]. Even the traditional homogeneous scheduling problems are NP-hard [9], so heterogeneity makes this problem even harder, thus allowing exact methods being only useful for solving instances of reduced size.

When dealing with large problem instances, as demanded by the size of current HC systems, heuristic and metaheuristic methods [14,18,19] are the only viable options in practice to compute efficient schedules in reasonable execution times. In this context, the faster the scheduler, the quicker the HC system can allocate new tasks and, as a consequence, the better its utilization degree (and the corresponding income). However, it is very hard, particularly for metaheuristics, to meet the wall-clock time constraint imposed in current heterogeneous cluster computing infrastructures and in grid computing systems. In fact, scheduling dozens of thousands of tasks is even slow for basic low level heuristic methods, which are usually much faster than metaheuristics. This work is focussed on these latter heuristic methods and the use of parallelism to reduce their computational times. The actual scientific contribution therefore lies in the parallelization of a stochastic local search (rPALS [15]) on GPU (Graphic Processing Units) cards. It has been called gPALS. Its main goal is to profit from the computing power of these newly available massively parallel platforms in order to address very large HCSP instances. Indeed, the testbed used is composed of problem instances that range from 8096 to 32768 tasks, and 256 to 1024 machines, respectively. Averaging over 60 different instances, the results have shown that, compared to the state-of-the-art deterministic MinMin heuristic [11], gPALS is able to reach task schedules with a 5% lower makespan more than 11 times faster.

The rest of the manuscript is organized as follows. The next section presents the HCSP formulation and briefly describes the list scheduling heuristics used for initializing the proposed gPALS and in the results comparison. Section 3 introduces the main concepts about GPU computing. The details about the GPU implementation for the stochastic local search method proposed to efficiently solve the HCSP are presented in Section 4. The experimental analysis is described in Section 5, studying the numerical efficacy and the computational efficiency of the proposed method in a number of large-sized HCSP scenarios. Finally, Section 6 presents the main conclusions of the research and formulates the main lines for future work.

2 Heterogeneous computing scheduling

This section introduces the HCSP formulation and presents some considerations about the execution time estimation model used in the problem instances to solve. In addition, the classic deterministic heuristics applied for initializing the proposed gPALS method and the one used as a baseline to compare the gPALS results are described.

2.1 Formal definition

An HC system is a computational platform composed of many computational resources, also called *processors* or *machines*. The scheduling problem in HC considers a set of tasks with variable computing demands to be executed on the system. A task is the atomic unit of workload, so it cannot be divided into smaller chunks, nor interrupted after it is assigned to a machine (i.e., the scheduling problem follows a *non-preemptive* model). The execution times of each task vary from one machine to another, so there will be competition among tasks for using those machines able to execute them in the shortest time.

The most usual objective to minimize in scheduling is the *makespan*, defined as the time spent from the moment when the first task begins its execution to the moment when the last task is completed. However, many other objectives have been considered in scheduling problems [12].

The following formulation presents the mathematical model for the HCSP:

- given an HC system composed of a set of machines $M = \{m_1, m_2, \ldots, m_L\}$ and a collection of tasks $T = \{t_1, t_2, \ldots, t_N\}$ to be executed on the system,
- let there be an *execution time function* $ET : T \times M \to \mathbf{R}^+$, where $ET(t_i, m_j)$ is the time required to execute the task t_i in the machine m_j,
- the goal of the HCSP is to find an assignment of tasks to machines (a function $f : T^N \to M^L$) which minimizes the *makespan* metric, defined in Eq. 1.

$$\max_{m_j \in M} \sum_{\substack{t_i \in T: \\ f(t_i) = m_j}} ET(t_i, m_j) \ . \quad (1)$$

Using the 3-field notation from Graham *et al.* [10], the HCSP is denoted $Rm|1|Cmax$.

2.2 Execution time estimation model

In this work, we adopted the *expected time to compute* (ETC) performance estimation model by Ali *et al.* [2], which has been widely used by the research community when facing the HCSP. ETC provides an estimation for the execution time of a collection of tasks in an HC system, taking into account three key properties: machine heterogeneity, task heterogeneity, and consistency.

Machine heterogeneity evaluates the variation of execution times for a given task across the HC resources. A system with similar computing resources has low machine heterogeneity, while high machine heterogeneity represents HC systems with computing resources of different power. *Task heterogeneity* represents the variation of the tasks execution times for a given machine. In a high task heterogeneity scenario, different types of applications are submitted to execution, from simple programs to complex tasks which require large CPU times to be performed. On the other hand, low task heterogeneity models those scenarios when the tasks computational requirements, and thus their execution times, are similar for a given machine.

The ETC model considers a second classification. In a *consistent* ETC scenario, whenever a given machine m_j executes any task t_i faster than other machine m_k, then machine m_j executes all tasks faster than machine m_k. An *inconsistent* ETC scenario lacks of structure among the computing demands of tasks and the computing power of machines, so a given machine m_j may be faster than another machine m_k when executing some tasks, and slower for others. In addition, a third category of *semi-consistent* ETC scenarios is included, to model those inconsistent systems that include a consistent subsystem.

2.3 List scheduling heuristics

Several deterministic heuristics have been proposed for HC scheduling. One of the most used class of such methods is list scheduling heuristics [11]. List scheduling methods work by assigning priorities to tasks based on a particular criteria, sorting the list of tasks in decreasing priority, and assigning each task to a processor, regarding both the task priority and the processor availability.

Variations of two well-known list scheduling heuristics have been used in this work to generate the initial solution for the gPALS method:

- *Minimum Completion Time* (MCT) considers the set of tasks sorted in an arbitrary order. Then, it assigns each task to the machine with the minimum ET for that task.
- *MinMin* greedily picks the task that can be completed the soonest. The method starts with a set U of all *unmapped* tasks, calculates the MCT for each task in U for each machine, and assigns the task with the minimum overall MCT to the machine that executes it faster. The mapped task is removed from U, and the process is repeated until all tasks are mapped.

The MinMin heuristic provides an excellent packing of tasks for HC environments with high level of heterogeneity of both tasks and machines, thus computing better makespan values than other well-known list scheduling heuristics [11]. For this reason, the MinMin results are used in this work as a reference baseline for comparing the results computed with the proposed local search algorithm.

3 GPU computing

GPUs were originally designed to exclusively perform the graphic processing in computers, allowing the Central Process Unit (CPU) to concentrate on the remaining computations. Nowadays, GPUs have a considerably large computing power, provided by hundreds of processing units with reasonable fast clock frequencies. In the last ten years, GPUs have been used as a powerful parallel hardware architecture to achieve efficiency in the execution of applications.

GPU programming and CUDA. The first GPUs used for general-purpose computing were programmed using low-level mechanisms such as the interruption services of the BIOS, or by using graphic APIs such as OpenGL and DirectX [6].

Later, the programs for GPU were developed in assembly language for each card model, and they had very limited portability. So, high-level languages were developed to fully exploit the capabilities of the GPUs. In 2007, NVIDIA introduced CUDA (Compute Unified Device Architecture) [16], a software architecture for managing the GPU as a parallel computing device without requiring to map the data and the computation into a graphic API.

CUDA extends the C language, and it is available since cards of the GeForce 8 Series onwards. Three software layers are used in CUDA to communicate with the GPU (see Fig. 1): a low-level hardware driver that performs the CPU-GPU data communications, a high-level API, and a set of libraries such as CUBLAS for linear algebra and CUFFT for Fourier transforms calculation.

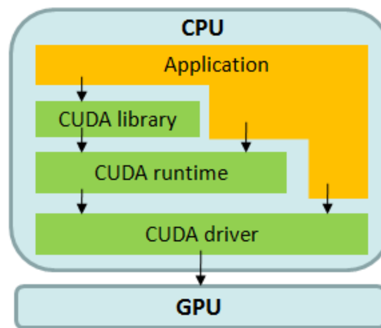


Fig. 1. CUDA architecture.

For the CUDA programmer, the GPU is a computing device which is able to execute a large number of threads in parallel. A specific procedure to be executed many times over different data can be isolated in a GPU-function using many execution threads. The function is compiled using a specific set of instructions and the resulting program (named *kernel*) is loaded in the GPU. The GPU has its own DRAM, and the data are copied from the DRAM of the GPU to the RAM of the host (and viceversa) using optimized calls of the CUDA API.

The CUDA architecture is built around a scalable array of multiprocessors, each one having eight scalar processors, one multithreading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with reduced overhead. The threads are grouped into *blocks* (with up to 512 threads), which are executed in a single multiprocessor of the graphic card, and the blocks are grouped in *grids*. Each time that a CUDA program calls a grid to be executed in the GPU, each of the blocks in the grid is numbered and distributed to an available multiprocessor. When a multiprocessor receives a block to be executed, it splits the threads into *warps*, a set of 32 consecutive threads. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Otherwise, the warp serializes the threads. When a block finishes its execution, a new block is assigned to the available multiprocessor.

The threads are able to access the data using three memory spaces: a *shared memory* which can be used by the threads in the block; the *local memory* of the thread; and the *global memory* of the GPU. Minimizing the access to the slower memories (the local memory of the thread and the global memory of the GPU) is a very important feature to achieve high efficiency in GPU computing. On the other hand, the shared memory is placed within the GPU chip, thus providing a faster way to store data, such as the registers of each multiprocessor.

4 gPALS: a GPU implementation of a stochastic local search scheduler

This section describes both rPALS, the base algorithm that has been parallelized, and gPALS, its deployment on GPU cards.

4.1 rPALS

The stochastic local search algorithm proposed in this work to efficiently solve the HCSP is based on PALS [1]. The original PALS method is a deterministic local search algorithm originally proposed for the DNA fragment assembly problem.

PALS works on a single solution s, which is iteratively modified by applying a series of movements aimed at locally improving their objective function value $f(s)$. The movement operator performs a modification on two positions i and j in the solution s, while the key step is the calculation of the objective function variation $\Delta f_{(i,j)}$ when applying a certain movement. When the calculation of $\Delta f_{(i,j)}$ can be performed without significantly increasing the computational cost of the algorithm, PALS provides a very efficient search pattern for combinatorial optimization problems.

In this work, a randomized variant of PALS (rPALS), has been used for the HCSP [15]. The aim of the algorithm is to reach accurate schedules in very short times. To do so, from a initial solution computed by a fast scheduling heuristic, rPALS iteratively applies two basic operations that either swap or move randomly chosen tasks allocated to randomly chosen machines, thus avoiding exploring all the possible neighbors. The algorithm has been designed under the paradigm of simplicity; by using simple search operators, the resulting rPALS method is able to scale up in order to face realistic medium-sized HCSP instances.

4.2 gPALS

The emergence of general purpose GPU computing has opened new research lines specially promising in this field of scheduling and planning. Indeed, this new technology will help to address more and more larger problem instances (up to 32768 tasks and 1024 machines in this work) in reasonable wall-clock times which are closer to the actual HC infrastructures. The key issue is to fully exploit the massively parallelism of the GPU cards. This is precisely the main design goal of gPALS, the GPU version of rPALS.

Algorithm 1 presents the pseudo-code of the gPALS algorithm for the HCSP. The method starts by generating an initial schedule s using a list scheduling heuristic (e.g. MCT, pMin-Min/DD, etc.). Then, a search is performed in the GPU in order to find candidate movements to improve the schedule, this search is detailed in Algorithm 2. The GPU search returns the best `GPU_BLOCKS` movements found. The best movement is always applied, the remaining movements are applied in random order as long as they do not modify a machine already modified by a previous movement, otherwise the movement is discarded. Once all the movements are either applied or discarded, the stopping criterion is tested and the algorithm either ends or performs a new iteration.

The movement search on the GPU is organized in blocks; there are `GPU_BLOCKS` blocks, each block having `GPU_THREADS` threads. Each block performs an independent local search in a randomly selected neighbourhood, with the threads in the block collaborating with each other to find the best movement in the assigned neighbourhood. Algorithm 2 presents the movement search performed on the GPU. First, each block deterministically selects a movement type (i.e. `MOVE` or `SWAP`). Then each thread in the block randomly selects the source and destination elements to modify. Each thread evaluates its assigned movement and computes a score for it. After each thread in the block evaluated its assigned movement, the best movement (i.e. the one with the lower score) is selected and returned to Algorithm 1 as the best movement found in the block.

Algorithm 1 gPALS for the HSCP

1: $s \leftarrow$ initialize using a list scheduling heuristic
2: **while** `STOPPING_CONDITION` is not met **do**
3: $m \leftarrow$ Parallel execution of the movement search kernel in s with 128 blocks with 256 threads each {A total of 32768 threads are launched}
4: $s \leftarrow$ Apply the best movement from m
5: $s \leftarrow$ Apply the rest of the movements in m in random order
6: **end while**
7: **return** s

As it can be seen, gPALS requires an initial solution which is iteratively improved (line 1 in Algorithm 1). For this initial solution to be generated, any classical list heuristic could be used in order to provide gPALS with a rather high quality task schedule. Two different versions of gPALS have been devised depending on this heuristic:

- gPALS$_{MCT}$: it uses MCT for generating the initial task schedule.
- gPALS$_{MMDD}$: the *pMin-Min/DD* (or parallel *Min-Min* with domain decomposition) heuristic is adopted. It is a multithreading version of the *Min-Min* algorithm, which performs a task domain decomposition and does not require any synchronization mechanism (see [13] for the details).

A diagram of the parallel model used in gPALS is presented in Fig. 2.

Algorithm 2 Movement search kernel for the GPU.

1: **shared** $M \leftarrow \emptyset$ {shared across all threads in the block}
2: $s \leftarrow$ Current schedule
3: movement $\leftarrow$ Choose which movement to perform
4: **if** movement is `TASK_SWAP` **then**
5: $t_x, t_y \leftarrow$ Choose two random tasks ($t_x \neq t_y$)
6: $m \leftarrow$ Swap t_x with t_y
7: **else if** movement is `TASK_MOVE` **then**
8: $t_x \leftarrow$ Choose a random task
9: $m_y \leftarrow$ Choose a random machine
10: $m \leftarrow$ Move t_x to m_y
11: **end if**
12: **if** makespan of the schedule s increases after the movement m **then**
13: $score \leftarrow \infty$
14: **else**
15: $ct_x \leftarrow$ Compute time of the machine m_x to which t_x is assigned
16: $ct'_x \leftarrow$ Compute time of the machine m_x after applying the movement
17: $ct_y \leftarrow$ Compute time of the machine m_y (when swapping, the machine to which t_y is assigned)
18: $ct'_y \leftarrow$ Compute time of the machine m_y after applying the movement
19: $score \leftarrow (ct'_x - max(ct_x, ct_y)) + (ct'_y - max(ct_x, ct_y))$
20: **end if**
21: $M \leftarrow M \cup m$
22: **synchronize()** {threads in the block}
23: $m_{best} \leftarrow$ Parallel reduce M to find best movement in the block
24: **return** m_{best}

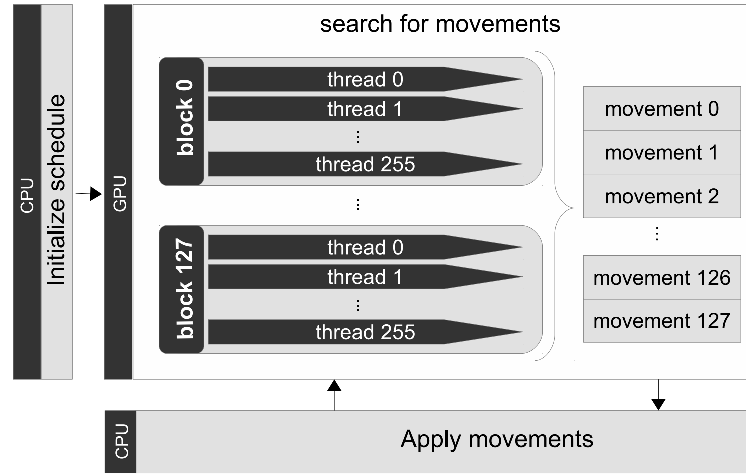


Fig. 2. Parallel model applied in gPALS.

5 Experimental analysis

This section introduces the set of HCSP instances and the computational platform used to evaluate the proposed LS algorithm. After that, the experiments conducted to determine the best values for the randomized PALS parameters are presented. Finally, the results obtained when solving realistic HCSP instances are analyzed in detail.

5.1 HCSP instances

To evaluate the proposed gPALS method, a specific set of 60 HCSP instances was used. These instances were randomly generated following the *range based* methodology proposed by Ali *et al.* [2], and they were previously employed to evaluate a cellular genetic algorithm scheduler for HC systems in the work by Pinel et al. [17].

The HCSP instances solved in this article model realistic large-sized HC infrastructures. Three problem dimensions were studied in the experimental analysis of gPALS: (tasks×machines) 8192×256, 16384×512, and 32768×1024. This dimensions are far more larger than the ones usually tackled in the related literature. For each problem dimension considered, 20 instances were used, follow the parametrization values suggested by Braun *et al.* [3].

5.2 Development and execution platform

The rPALS algorithm was implemented in C, using the standard `stdlib` library and compiled with gcc 4.1.2. The experimental analysis was performed in a Dell PowerEdge with QuadCore Xeon E5430 processor at 2.66 GHz, 8 GB RAM, and CentOS Linux (platform website: `http://www.fing.edu.uy/cluster`).

5.3 Results and discussion

This section is aimed at presenting the experimental results obtained. It has been structured in two separate subsections so as to analyze, firstly, the quality of the tasks schedules reached by MinMin, gPALS$_{MCT}$, and gPALS$_{MMDD}$ in terms of their makespan and, secondly, the parallel performance of the proposed approaches, gPALS$_{MCT}$ and gPALS$_{MMDD}$, with respect to MinMin.

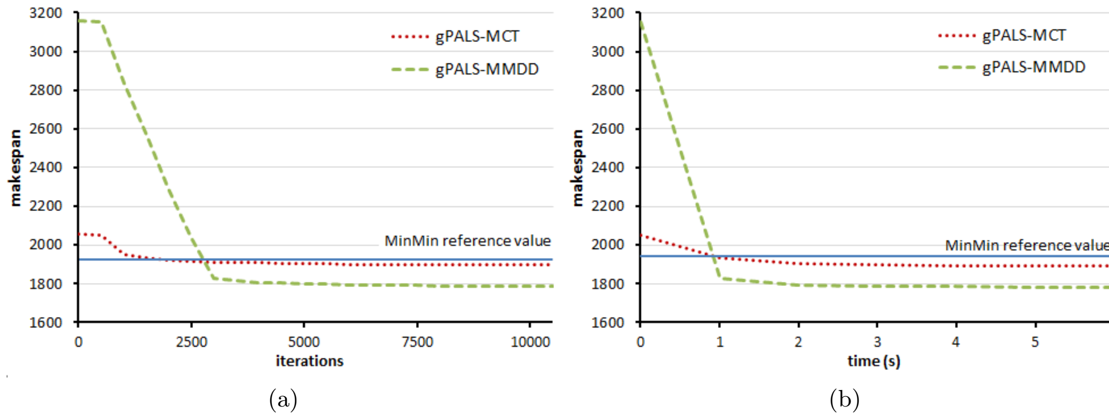
Numerical efficiency. Table 1 reports the makespan reached by MinMin and the two versions of gPALS for the three set of instances with increasing size. Before going into details, we want to make clear the experimental conditions. Whereas MinMin has been let to execute until it schedules all the tasks, i.e., until a full solution is built (it is a constructive heuristic), both gPALS$_{MCT}$ and gPALS$_{MMDD}$ stop when 30 seconds of GPU computation have elapsed (the loading time of the instance is not considered here). The cells with the best makespan are marked with a gray background.

Table 1. Makespan of the three algorithms for 60 HCSP instances, 20 for each dimension —8192×256, 16384×512, and 32768×1024.

	8192×256			16384×512			32768×1024		
	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$
1	1845.2	1835.4	1710.4	1934.1	1886.1	1777.0	1996.2	1927.3	1831.3
2	1889.9	1863.2	1740.0	1940.5	1889.9	1781.2	1979.0	1918.0	1824.3
3	1894.3	1831.0	1716.0	1949.0	1895.7	1782.1	1980.7	1919.7	1821.6
4	1890.1	1866.4	1743.0	1922.0	1887.1	1778.4	1982.8	1929.3	1830.6
5	1859.6	1843.7	1724.8	1904.1	1867.7	1757.1	1971.9	1918.5	1818.4
6	1863.4	1829.0	1715.4	1901.7	1885.7	1772.6	1973.4	1912.0	1816.3
7	1897.3	1862.3	1736.9	1945.5	1898.1	1787.6	1991.0	1926.0	1829.0
8	1874.5	1852.6	1738.4	1903.8	1878.6	1768.5	1991.8	1922.5	1822.0
9	1871.5	1853.3	1730.9	1937.3	1894.7	1782.4	1994.8	1929.5	1832.2
10	1865.9	1867.2	1742.5	1935.7	1877.6	1769.9	1997.1	1922.1	1822.9
11	1840.7	1823.8	1711.9	1937.4	1899.2	1786.7	1975.8	1914.7	1816.2
12	1867.3	1836.7	1724.2	1916.2	1871.6	1762.8	1974.2	1912.2	1813.5
13	1895.4	1867.5	1744.6	1911.8	1884.8	1771.6	1978.6	1915.2	1818.6
14	1884.8	1841.8	1725.4	1927.6	1898.7	1784.0	1988.1	1923.3	1824.3
15	1851.0	1828.2	1710.8	1944.4	1901.0	1787.5	1972.1	1915.8	1819.3
16	1846.3	1837.2	1724.1	1939.5	1886.5	1777.9	1979.3	1913.3	1814.2
17	1874.7	1818.1	1707.8	1933.6	1878.4	1764.9	1991.8	1916.7	1814.9
18	1862.8	1856.5	1736.7	1929.5	1887.0	1776.4	1986.8	1922.5	1825.5
19	1892.5	1853.4	1732.7	1910.8	1880.6	1765.6	1975.2	1914.5	1814.2
20	1869.0	1853.8	1731.2	1941.0	1891.6	1781.0	1991.7	1919.9	1819.2

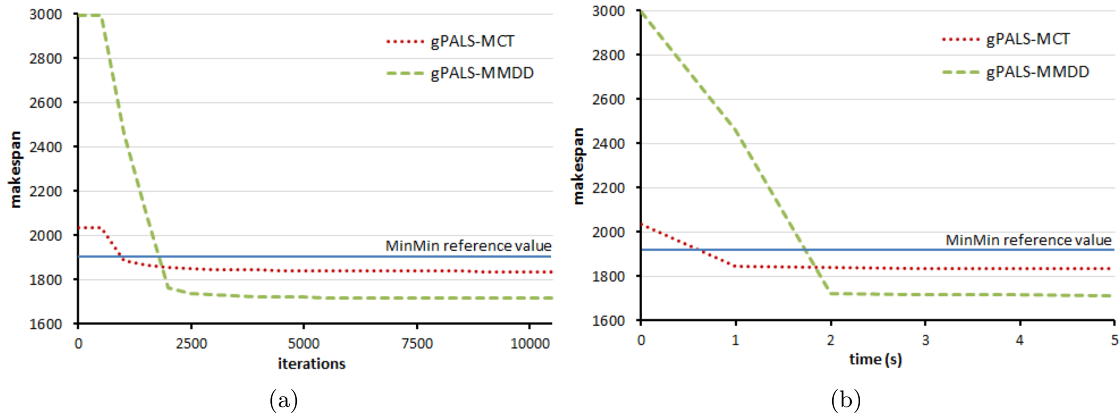
The experimental results in Table 1 clearly point out that gPALS$_{MMDD}$ is the algorithm that reached the task schedules that most reduces the makespan for all the instances addressed. This occurs consistently for the three instances sizes, i.e., 8192×256, 16384×512, and 32768×1024. Averaging over all the instances of the same size, gPALS$_{MMDD}$ improves the makespan computed by MinMin in 7.72%, 7.91%, and 8.18%, respectively. It is important to note the relevance of these values, given the experimental conditions. Though slightly, these average values show that, the larger the instances, the better the improvement, and this has been achieved by keeping the same computation time, i.e., 30 seconds. That is, for search spaces very much larger (both the number of tasks and machines is doubled), our approach is able to improve even more MinMin, which requires in turn very much longer execution times (see the next section). To a lesser extent, the same claims hold for gPALS$_{MCT}$: the average improvements are also increasing with the instance size, but only 1.37%, 2.13%, and 3.24%, respectively.

In order to better support our claims, Fig. 3 displays the evolution of the makespan of a typical 8192×256 instance in terms of (a) the iterations and (b) the execution time of gPALS$_{MCT}$ and gPALS$_{MMDD}$ in a typical execution, respectively. The makespan obtained by Min-Min is also included as a baseline for the comparison. These subfigures shows a very interesting fact. For gPALS, the more accurate the initial solution (in this case, that computed by MCT), the earlier the stagnation in a local minimum. Indeed, it can be seen that pMin-



(a) (b)

Fig. 3. Evolution of the makespan during a typical execution of the three compared algorithms for a 8192×256 instance with respect to (a) the iterations of gPALS and (b) its wall-clock time.



(a) (b)

Fig. 4. Evolution of the makespan during a typical execution of the three compared algorithms for a 16384×512 instance with respect to (a) the iterations of gPALS and (b) its wall-clock time.

MinDD reaches a task schedule with much higher (worse) makespan, and then gPALS is able to iteratively move and swap tasks between machines that allow the makespan to be continuously reduced up to the iteration 2500. With respect to Min-Min, gPALS$_{MCT}$ requires around 1000 generations to reach a lower makespan, whereas gPALS$_{MMDD}$ is around iteration 2000. If we now turn to analyze the evolution with respect the execution time, the picture changes. The first remark here is that the two gPALS versions outperform MinMin after just one single second of computation, clearly showing their suitability for addressing this large instances of the HCSP problem. The second remark raises when comparing gPALS$_{MCT}$ and gPALS$_{MMDD}$: the latter also requires just one second to reach a more accurate task schedule than the former.

Figure 4 presents the evolution of makespan values with respect to both the iterations of gPALS and the execution time, but for a representative 16384×512-sized instance. All the previous claims hold as well with the only difference in subfigure (b), in which now the generation of the initial solution for $gPALS_{MMDD}$ with the pMin-MinDD heuristic takes longer and delays outperforming both MinMin and $gPALS_{MCT}$ about one second. The same behavior was detected for the other HCSP instances in the benchmark set solved in this article.

Table 2. Wall-clock of the three algorithms (in seconds) for 60 HCSP instances, 20 for each dimension —8192×256, 16384×512, and 32768×1024.

	8192×256			16384×512			32768×1024		
	MinMin	$gPALS_{MCT}$	$gPALS_{MMDD}$	MinMin	$gPALS_{MCT}$	$gPALS_{MMDD}$	MinMin	$gPALS_{MCT}$	$gPALS_{MMDD}$
1	15.0	11.6	10.6	110.7	9.7	16.7	839.8	21.1	112.6
2	15.0	9.8	9.8	110.6	9.9	17.0	838.8	20.5	115.8
3	14.8	9.7	8.5	110.6	9.7	17.6	842.5	18.9	98.5
4	14.9	10.4	9.7	110.7	11.4	19.1	841.7	20.9	111.3
5	14.9	8.1	8.4	111.2	13.1	17.1	845.3	22.2	105.7
6	14.9	7.9	8.9	111.4	12.0	17.9	834.6	21.9	110.9
7	14.9	8.4	9.0	111.3	11.7	19.0	837.4	20.6	105.5
8	15.3	8.0	8.4	111.3	12.4	17.1	843.2	19.6	112.3
9	15.0	8.6	8.3	111.1	10.0	18.1	838.4	19.6	105.0
10	15.0	11.8	8.2	110.7	12.4	18.9	839.3	21.4	122.2
11	14.8	8.8	8.6	110.8	13.1	19.8	840.4	20.6	109.6
12	14.9	7.9	8.2	110.9	12.6	19.8	838.7	19.1	100.3
13	14.9	7.7	8.2	110.7	13.4	19.9	843.1	20.3	110.3
14	14.9	7.5	8.2	110.8	13.5	20.2	841.8	19.9	112.1
15	14.9	7.7	8.2	110.5	13.4	19.1	844.0	20.8	107.4
16	15.1	8.7	8.2	110.8	12.7	19.9	834.9	22.0	111.2
17	15.0	7.5	8.2	111.3	13.3	19.9	837.6	21.1	106.1
18	14.6	9.0	8.3	111.3	13.2	19.7	842.7	20.5	108.3
19	14.9	7.5	8.2	111.2	13.0	20.2	838.8	19.3	125.0
20	14.9	8.1	8.2	111.1	13.3	19.1	840.0	19.9	99.3

Parallel performance. We have already provided the reader with some hints about the main features of the computational times of the three algorithms, but we now want to detail them in a separate experimentation. Table 2 includes the wall-clock time of MinMin, $gPALS_{MCT}$ and $gPALS_{MMDD}$ for the 60 HSCP instances with increasing size considered in this work. The experimental conditions for the two gPALS methods have changed: they stop when they reach a task schedule with a lower makespan than that of MinMin (in the previous section, the stopping condition was to reach 30 seconds of GPU computation).

The first clear claim is that the two gPALS versions are always faster than MinMin to achieve an task schedule with the same makespan. The truly interesting point here is that, the larger the instance, the higher the reduction in the

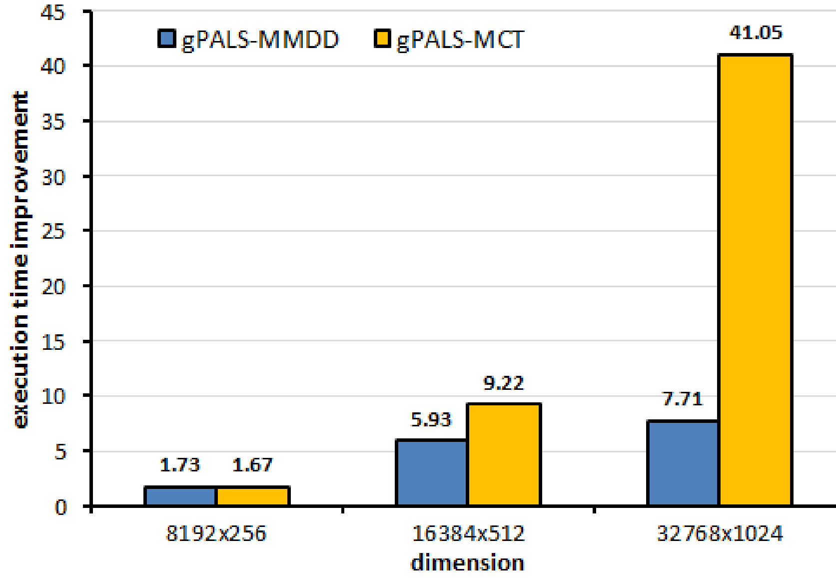


Fig. 5. Average execution time improvements of gPALS$_{MCT}$ and gPALS$_{MMDD}$ with respect to MinMin.

execution times. Indeed, MinMin requires roughly 15, 110, and 840 seconds to build a solution for 8192×256, 16384×512, and 32768×1024 instances, respectively, whereas gPALS$_{MCT}$ and gPALS$_{MMDD}$ need 8, 12, and 20 seconds, and 9, 18, and 110 seconds, respectively. These differences can be clearly seen in Fig. 5, which displays the average execution time improvements over all the instances of the same size reached by gPALS$_{MCT}$ and gPALS$_{MMDD}$. The execution time improvement refers to the reduction in the execution time of an algorithm that runs in a parallel computing platform (in our case the GPU) with respect another one that executes sequentially, i.e., $\frac{t_{CPU}}{t_{GPU}}$. For the smaller instance considered in this work, the two gPALS approaches perform the same, with execution time improvements of 1.73 and 1.67. However, as long as the size of instance increases, MinMin requires more time to complete, i.e., it does not scale well, whereas our approaches do scale properly, specially gPALS$_{MCT}$, which has been able to reach an execution time improvement of 40.1 for the largest instance. We would like to dive a little bit more on the results of the two gPALS versions and explain why the execution time improvements of gPALS$_{MCT}$ is much higher. Obviously, it has to do with the computational time of the initial task schedule by the heuristic. MCT is a extremely fast method whose translation to the GPU does not make sense because little benefits would be obtained. On the other hand, pMinMinDD is much heavier and, even ported to the GPU, takes longer to build a solution, what reduces its execution time improvements.

6 Conclusions

This work has presented gPALS, a GPU implementation of a randomized local search procedure for addressing large instances of the scheduling problem in HC systems. The aim of the algorithm is to reach accurate schedules in very short times for large HCSP instances using the parallel computing resources available in GPUs. To do so, from a initial solution computed by either the MCT heuristic or a parallel version of the MinMin heuristic, two variants of gPALS were implemented in GPU by iteratively applying two basic operations that either swap or move randomly chosen tasks allocated to randomly chosen machines. The key is that the variation in the makespan of these operations can be computed efficiently and, consequently, several millions operations can be performed in few seconds.

The experimental analysis performed using a testbed with 60 large HCSP instances of three increasing dimensions (up to 32768 tasks and 1024 machines) compared the two proposed versions of gPALS against Min-Min, one of the best state-of-the-art list scheduling heuristics for HC environments. The experimental results demonstrate that the proposed gPALS implementations are able of compute better makespan values than MinMin in all the 60 studied instances.

The gPALS$_{MMDD}$ variant is the best method between the two GPU implementations, obtaining significant improvements (up to 8.18%) with respect to MinMin. These reductions in the makespan obtained by gPALS$_{MMDD}$ have taken a wall-clock time of 30 seconds, which represent a factor of almost 8$\times$ in the computational efficiency with respect to the MinMin scheduler. On the other hand, gPALS$_{MCT}$ computed schedules significantly faster than both gPALS$_{MMDD}$ and MinMin, achieving execution time improvements up to 41.05 with respect to MinMin. The solution computed by gPALS$_{MCT}$ improves upon the ones computed using MinMin, but they have lower quality (i.e., larger makespan values) than those found by the gPALS$_{MMDD}$ implementation. Regarding the execution time comparison, both gPALS implementations are able to improve over the MinMin makespan result in only a few seconds of execution time (without counting the time spent in computing the initial solution).

The previously commented results have demonstrated that the new gPALS$_{MMDD}$ algorithm is an accurate and very efficient scheduler for the HCSP instances tackled in this article.

Two main lines are proposed for future work: improve the efficacy of the search in gPALS, and also to enhance the computational efficiency of the proposed optimization method. Regarding the first issue, we propose to analyze carefully the landscape of the HCSP, in order to design specialized basic operations that further improve the efficacy of the search in gPALS, by avoiding to explore non-promising regions of the search space. On the other hand, in order to be able to address even larger HCSP instances in shorter times, we also plan to engineer a more efficient version of gPALS by employing domain-decomposition parallel computing techniques in GPU.

Acknowledgments. The work of S. Iturriaga and S. Nesmachnow has been partially supported by ANII and PEDECIBA, Uruguay. The work of F. Luna and E. Alba has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contracts TIN2008-06491-C04-01 (the MSTAR project) and TIN2011-28194 (the roadME project), and by the Andalusian Government under contract P07-TIC-03044 (the DIRICOM project).

References

1. E. Alba and G. Luque. A new local search algorithm for the DNA fragment assembly problem. In C. Cotta and J. van Hemert, editors, *Proceedings of 7th European Conference on Evolutionary Computation in Combinatorial Optimization*, volume 4446 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.
2. S. Ali, H. Siegel, M. Maheswaran, S. Ali, and D. Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proceedings of the 9th Heterogeneous Computing Workshop*, page 185, Washington, DC, USA, 2000. IEEE Computer Society.
3. T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. 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):810–837, 2001.
4. H. El-Rewini, T. Lewis, and H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
5. M. Eshaghian. *Heterogeneous Computing*. Artech House, Norwood, MA, USA, 1996.
6. R. Fernando, editor. *GPU gems*. Addision-Wesley, Boston, 2004.
7. I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.
8. R. Freund, V. Sunderam, A. Gottlieb, K. Hwang, and S. Sahni. Special issue on heterogeneous processing. *Journal of Parallel and Distributed Computing*, 21(3), 1994.
9. M. Garey and D. Johnson. *Computers and intractability*. Freeman, 1979.
10. R. Graham, J. Lawler, E. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann of Discrete Mathematics*, 5:287–326, 1979.
11. Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computer Surveys*, 31(4):406–471, 1999.
12. J. Leung, L. Kelly, and J. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.
13. S. Nesmachnow and M. Canabé. GPU implementations of scheduling heuristics for heterogeneous computing environments. In *Proceedings of the XVII Congreso Argentino de Ciencias de la Computación*, 2011.
14. S. Nesmachnow, H. Cancela, and E. Alba. A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. *Applied Soft Computing*, 12(2):626–639, 2012.
15. S. Nesmachnow, F. Luna, and E. Alba. An efficient stochastic local search for heterogeneous computing scheduling. In *15th International Workshop on Nature Inspired Distributed Computing*, 2012.

16. nVidia. CUDA website. Available online http://www.nvidia.com/object/cuda_home.html, 2010. Accessed on July 2011.
17. F. Pinel, B. Dorronsoro, and P. Bouvry. Solving very large instances of the scheduling of independent tasks problem on the GPU. *Journal of Parallel and Distributed Computing*, 2012. In press, DOI:10.1016/j.jpdc.2012.02.018.
18. F. Pinel, J. Pecero, P. Bouvry, and S. U. Khan. A two-phase heuristic for the scheduling of independent tasks on computational grids. In *2011 International Conference on High Performance Computing and Simulation (HPCS)*, pages 471 – 477, 2011.
19. G. Ritchie and J. Levine. A fast, effective local search for scheduling independent jobs in heterogeneous computing environments. In *Proceedings of the 22nd Workshop of the UK Planning and Scheduling Special Interest Group*, 2003.