A New Parallel Asynchronous Cellular Genetic Algorithm for Scheduling in Grids

Frédéric Pinel, Bernabé Dorronsoro, and Pascal Bouvry
Faculty of Science, Technology, and Communications
University of Luxembourg
{*frederic.pinel, bernabe.dorronsoro, pascal.bouvry*}*@uni.lu*

Abstract

We propose a new parallel asynchronous cellular genetic algorithm for multi-core processors. The algorithm is applied to the scheduling of independent tasks in a grid. Finding such optimal schedules is in general an NP-hard problem, to which evolutionary algorithms can find near-optimal solutions. We analyze the parallelism of the algorithm, as well as different recombination and new local search operators. The proposed algorithm improves previous schedules on benchmark problems. The parallelism of this algorithm suits it to bigger problem instances.

1. Introduction

Genetic algorithms (GAs) are population based algorithms that explore the search space by iteratively applying a set of stochastic operators on the solutions composing the population. Cellular genetic algorithms (CGAs) are a kind of GA with decentralized population that have demonstrated to outperform regular GAs in a large number of problems with different features and belonging to distinct domains [1]. In CGAs, the number of individuals that can mate to a given one is restricted to only those ones that are located next to it, i.e., to its neighborhood (as it is shown in Fig. 1). By adopting this simple idea, we achieve a slow spread of solutions through the population, and different regions of the population will therefore converge to different areas of the search space. The effect is that the population diversity is kept for longer while at the same time different niches appear in the population.

From the appearance of the first CGA by Robertson in 1987 [2], different parallel implementations have been proposed for CGAs in the literature (a complete survey on parallel CGAs can be found in [1]). From those designed for massively parallel computers (with SIMD architecture) having thousands of processors [3] to the ones recently proposed for clusters of computers by Luque et al. [4], [5]. However, very few parallel designs have been proposed for CGAs, despite their parallel nature. The reason is probably the high communication level required due to the tight relations among individuals. In this paper, we propose a new parallel CGA for multi-core processors. Due to the shared memory existing in this kind of architectures, the tight communications among individuals is less of a problem. Therefore, it allows us to take profit of the high performance of parallel implementations without the typical problems at the communication level existing in CGAs.

The main contributions of this paper are the application to task scheduling in grids [6] of a parallel CGA [7], and a new local search operator suited to this problem is proposed. The resulting algorithms are faster than the state-of-the-art meta-heuristics in the literature, providing better results.

This paper is organized as follows: Section 2 describes the problem of scheduling in grids. Section 3 presents the new asynchronous cellular genetic algorithm. In Section 4, we provide results, analyze the behavior of our algorithm, and compare it to other well-known and current state-of-the-art algorithms. Finally, we summarize our main conclusions and future works.

2. Batch Scheduling in Grids

Task scheduling in computational grids is a family of problems that capture the most important needs of grid applications for efficiently allocating tasks to resources in a global and dynamic environment. Therefore, several versions of the problem can be formulated according to the needs of such applications.

We give in Section 2.1 a description of the problem we are considering in this work, while Section 2.2 mathematically describes the problem we are optimizing.

2.1. Problem description

In this work, we consider a version of the problem that arises quite frequently in parameter sweep applications, such as Monte-Carlo simulations [8]. In these applications, many tasks with almost no interdependencies are generated and submitted to the grid system. In fact, more generally, the scenario in which the submission of independent tasks to a grid system is quite natural given that grid users independently submit their tasks to the grid 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 grid system. Indeed, it is a measure of its productivity (throughput).

In our scenario, the tasks originate from different users or applications. They have to be individually processed by a single resource unless it drops from the grid due to its dynamic environment (*non-preemptive* mode). They are independent of each other and could specify hardware and/or software requirements over resources. Also, resources could dynamically be added/dropped from the grid. They can process one task at a time, and have their own computing characteristics regarding the consistency of computing. More precisely, assuming that the computing time needed to perform a task is known (assumption that is usually made in the literature [9], [10], [11]), we use the Expected Time to Compute (ETC) model by Braun et al. [9] to formalize the instance definition of the problem as follows:

- A *number* of independent (user/application) *tasks* to be scheduled.
- A *number* of heterogeneous *machine* candidates to participate in the planning.
- The *workload* of each task (in millions of instructions).
- The *computing capacity* of each machine (in *mips*).
- Ready time $ready_m$ indicates when machine m will have finished the previously assigned tasks.
- The Expected Time to Compute (ETC) matrix ($nb\_tasks \times nb\_machines$) in which $ETC[t][m]$ is the expected execution time of task t on machine m.

2.2. Optimization criterion

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

$$\min_S \max\{F_t : t \in Tasks\} \ , \quad (1)$$

where F_t is the finishing time of task t in schedule S.

For a given schedule, it is quite useful to define 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:

$$completion[m] = ready_m + \sum_{t \in S^{-1}(m)} ETC[t][m] \ . \quad (2)$$

We can then use the values of completion times to compute the makespan as follows:

$$\min_S \max\{completion[m] \mid m \in Machines\} \ . \quad (3)$$

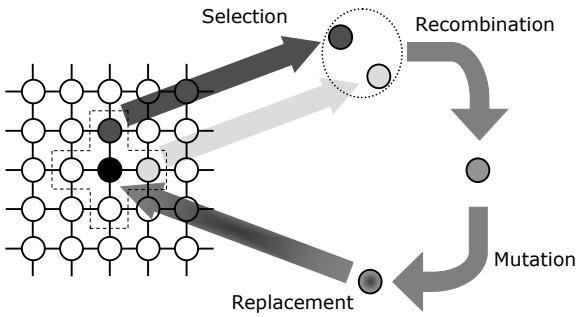

Figure 1: In cellular GAs, individuals are only allowed to interact with their neighbors.

3. The Proposed Algorithm

In this section, we present the parallel asynchronous CGA we used for this problem. First, Section 3.1 briefly introduces a regular canonical cellular GA. Then, Section 3.2 presents our proposed parallel approach. Finally, Section 3.3 describes our representation of the scheduling problem, and the new local search operator.

3.1. Cellular GAs

Cellular GAs [1], [3], [12] are structured population algorithms with a high explorative capacity. The individuals composing their population are (usually) arranged in a two dimensional toroidal mesh. This mesh is also called grid. Only neighboring individuals (i.e., the closest ones measured in Manhattan distance) are allowed to interact during the breeding loop (see Figure 1). This way, we introduce a form of isolation in the population that depends on the distance between individuals. Hence, the genetic information of a given individual spreads slowly through the population (since neighborhoods overlap). The genetic information of an individual will need a high number of generations to reach distant individuals, thus avoiding premature convergence of the population. By structuring the population in this way, we achieve a good exploration/exploitation trade-off on the search space. This improves the capacity of the algorithm to solve complex problems [1], [13].

Algorithm 1 Pseudo-code for a canonical CGA (asynchronous).

```
1: while ! StopCondition() do
2:   for all ind in population do
3:     neigh ← get_neighborhood(ind);
4:     parents ← select(neigh);
5:     offspring ← recombine(p_comb, parents);
6:     mutate(p_mut, offspring);
7:     evaluate(offspring);
8:     replace(ind, offspring);
9:   end for
10: end while
```

A canonical CGA follows the pseudo-code of Algorithm 1. In this basic CGA, Each individual in the grid is iteratively evolved (line 2). A generation is the evolution of all individuals of the population. Individuals may only interact with individuals belonging to their neighborhood (line 3), so parents are chosen among the neighbors (line 4) with a given criterion. Recombination and mutation operators are applied to the individuals in lines 5 and 6, with probabilities p_comb and p_mut, respectively. Afterwards, the algorithm computes the fitness value of the new offspring individual (or individuals) (line 7), and replaces the current individual (or individuals) in the population (line 8), according to a given replacement policy. This loop is repeated until a termination condition is met (line 1), for example: the total elapsed processing time or a number of generations.

The CGA described by Algorithm 1 is called asynchronous, because the population is updated with next generation individuals immediately after their creation. These new individuals can interact with those belonging to their parent's generation. Alternatively, we can place all the offspring individuals into an auxiliary population, and then replace all the individuals of the population, with those from the auxiliary population, at once. This last version is referred to as the synchronous CGA model. As it was studied in [1], [14], the asynchronous CGAs converge the population faster than the synchronous CGAs.

3.2. Parallel Asynchronous Cellular GA

In this paper, we present our parallel asynchronous CGA (PA-CGA), based on [7]. We partition the population into a number of contiguous blocks with a similar number of individuals (Figure 2). Each block contains $pop\_size/\#threads$ individuals, where $\#threads$ represents the number of concurrent threads executed. We partition the population by assigning successive individuals to the same block. The successor of an individual is its right neighbor. We move to the next row when we reach the end of a row (our grid is 2-dimensional). We assign each block to a different thread, which will evolve the individuals of its block.

In order to preserve the exploration characteristics of the CGA, communication between individuals of different blocks is made possible. As mentioned in the previous section, at each evolution step of an individual, we define its neighborhood. This neighborhood may include individuals from other population blocks. This allows an individual's genetic information to cross block boundaries.

Threads evolve their population block independently. They do not wait on the other threads to complete their generation (the evolution of all the individuals in their block) before pursuing their evolution. Hence, if a breeding loop takes longer for an individual of a given thread, the individuals evolved by the other threads may go through more generations.

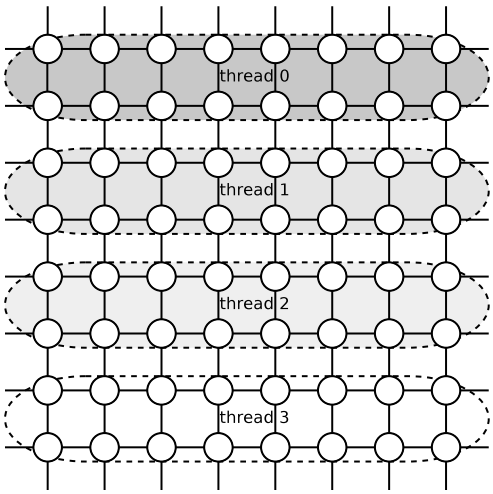

Figure 2: Partition of an 8x8 population over 4 threads.

The combination of a concurrent execution model with the neighborhoods crossing block boundaries leads to concurrent access to shared memory. For example, the neighbor n of an individual i might belong to a different block. The thread evolving this other block could be updating individual n precisely when the thread evolving i is accessing it. Without care, this can result in incorrect results. This error can occur when selecting a parent from a neighborhood (non-atomic read operation), or recombining (a non-atomic read operation of the parent) that belongs to another block and is currently being replaced (a non-atomic write operation). To enable safe concurrent memory access, we synchronize access to individuals with a POSIX [15] read-write lock. This high-level mechanism allows concurrent reads from different threads, but not concurrent reads with writes, nor concurrent writes. In the two latter cases, the operations are serialized.

Asynchronous CGAs can visit individuals in different orders [14]. In this work, all threads will sweep through their population in the same fixed order. This means that we are using the line sweep policy in every block. Note that this is not exactly the same as the line sweep policy typically used in asynchronous CGAs. In our case, all blocks are updated concurrently. We experimented different sweep orders for different blocks, in hope of limiting memory contention, but we did not notice any significant improvement in the algorithm's execution speed. We attribute this to the unpredictable nature of the thread's execution, while the alternative sweep policies per thread assumed a predictable, fixed, thread execution by the operating system.

Algorithms 2 and 3 provide a more detailed description of the algorithm. Function $do\_parallel(f, parm)$ means that $f(parm)$ is executed by all threads in parallel, but on different data items. All threads join before the next instruction.

Algorithm 2 Pseudo-code for our proposed parallel asynchronous CGA (PA-CGA).

1: $t_0 \leftarrow time()$; // record the start time
2: $pop \leftarrow setup\_pop()$; // initialize population
3: $par \leftarrow setup\_blocks(pop)$; // set parameters for all threads
4: $do\_parallel(initial\_evaluation, par)$; // each thread evaluates its block
5: $do\_parallel(evolve, par, t_0)$; // each thread evolves its block, see Algorithm 3

Algorithm 3 Pseudo-code for $evolve()$.

1: **while** $time() - t_0 \leq time$ **do**
2: **for all** ind in a thread's block **do**
3: $neigh \leftarrow get\_neighborhood(ind)$;
4: $parents \leftarrow select(neigh)$;
5: $offspring \leftarrow recombine(p\_comb, parents)$;
6: $mutate(p\_mut, offspring)$;
7: $H2LL(p\_ser, iter, offspring)$;
8: $evaluate(offspring)$;
9: $replace(ind, offspring)$;
10: **end for**
11: **end while**

Function $initial\_evaluation()$ computes the fitness of all individuals in the initial population. The stop condition for this grid scheduling problem is a wall clock time. The asynchronous model moves the stop condition verification into $evolve$.

From Algorithm 3, we notice that the thread checks the current time after evolving all the individuals of its block. This could let the thread run for longer than the allowed time. We accept this approximation since one generation of the entire block takes less than 6 ms in our experiments, while the time is expressed in tens of seconds. The evolution step also performs a local search operation. This operation is presented in the next subsection. We parameterize $H2LL()$, our local search, with a number of iterations $iter$, which sets the number of passes. Finally, $evaluate()$ computes the makespan of the schedule.

3.3. Local Search and Solution Representation

We also propose a new local search operator for the problem considered.

We refer to a machine's completion time as its *load*. The local search operator moves a task, randomly chosen, from the most loaded machine to a selected candidate machine (the most loaded machine's completion time defines makespan). The candidate machines are the N least loaded (N is a parameter). A candidate machine is selected if its new completion time, with the addition of the task moved, is the smallest of all the candidates. This new completion time must also remain inferior to the makespan. Algorithm 4 describes this operator.

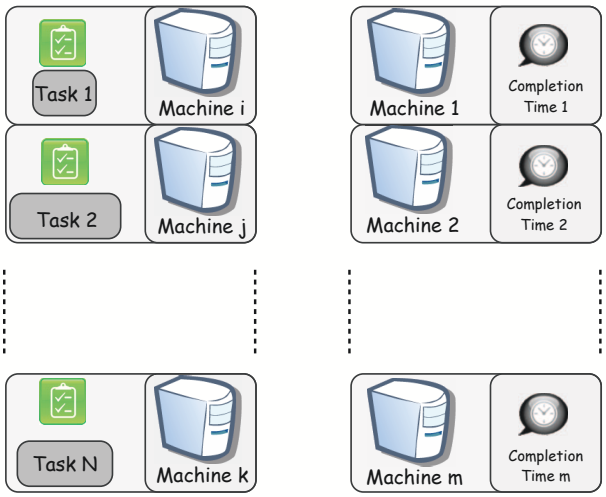

Figure 3: Representation of solutions. In addition to the task-machine assignments (left-hand side), we store the completion time for every machine too (right-hand side). Variation operators are only applied on the task-machine assignments.

The representation we use for independent task scheduling on grids is shown in Figure 3. It is composed of:

- an array S of integers, $S[t] = m$, representing the assignment of task t to machine m,
- an array CT of floating point values, $CT[m] = c$, representing the completion time of each machine m.

The completion times are often used, therefore maintaining up-to-date completion times for all machines speeds up computations. The $evaluate()$ function of Algorithm 3 only finds the maximum completion time. The completion times are kept up-to-date by each operator (recombine, mutation, local search). Such updates are efficiently performed by adding or removing the ETC of a task on a machine to the appropriate completion time. As can be noticed from Algorithm 4, we use the transposed ETC matrix. This increases the cache hit rate, and thus the overall performance of the algorithm. Indeed, when accessing an ETC for a task on a machine, this ETC value is cached, but so are the neighboring values (caches operate on cachelines). If we store the transposed ETC matrix, then these neighboring values are the ETC values for the next few tasks on the same machine (exactly how many depends on the size of a cacheline). So, if the schedule assigns one of the next tasks to the same machine, then this ETC value is present in cache. We measured an improvement in the algorithm's execution time of 5-10%. Indeed, this improvement is comparable to the uniform probability for such an event (next task assigned to same machine), $1/\#machines$, we use 16 machines in our experiments.

Algorithm 4 Pseudo-code for $H2LL$, our local search.

```
for all iter iterations do
  sort machines on ascending completion time
  task ← random task from last machines;
  best_score ← CT[last machines]; // makespan
  for all mac in pop_size/2 first machines do
    new_score ← CT[mac] + ETC[mac][task];
    if new_score < best_score then
      best_mac ← mac;
      best_score ← new_score;
    end if
  end for
  move task to best_mac if any
end for
```

4. Experiments

This section presents the results of our experiments with PA-CGA. Section 4.1 describes both the parameterization of the algorithm and the instances of the problem we are solving. Section 4.2 reports and discusses the results.

4.1. Parameters and Problem Instances

The algorithm parameters are summarized in Table 1. We are using a population of 256 individuals. The population is initialized randomly, except for one individual. The schedule for this individual results from the *Min-min* heuristic [16]. The linear 5 (L5) neighborhood, also called Von Neumann neighborhood, is composed of the 4 nearest individuals, plus the individual evolved. This neighborhood is chosen to reduce concurrent memory access. The 2 best neighbors are selected as parents. The recombination operators used are the one-point (opx) and the two-point crossover (tpx). The mutation operator moves one randomly chosen task to a randomly chosen machine. The newly generated offspring replaces the current individual if it improves the fitness value. Finally, the termination condition is an execution time of 90 seconds. The number of threads used in all our experiments ranges from 1 to 4. All threads run on one processor. The processor used for the experiments is an Intel Xeon E5440 clocked at 2.8 GHz, with 6 MB L2 cache. This processor has 4 cores. It was released in 2007.

Table 1: Parameterization of PA-CGA.

Population	16×16
Population initialization	Min-min (1 ind)
Cell update policy	fixed line sweep per block
Neighborhood	linear 5
Selection	best 2
Recombination	one-point and two-point crossover, $p\_comb = 1.0$
Mutation	move, $p\_mut = 1.0$
Local search	H2LL, $p\_ser = 1.0$, $iter = 5, 10$
Replacement	replace if better
Stopping criterion	90 seconds, wall time
Number of Threads	1 to 4

The benchmark instances consist of 512 tasks and 16 machines. These instances represent different classes of ETC matrices. The classification is based on three parameters: task heterogeneity, machine heterogeneity and consistency [17]. Instances are labelled as $u\_x\_yyzz.k$ where:

- u stands for uniform distribution (used in generating the matrix).
- x stands for the type of consistency (c for consistent, i for inconsistent, and s for semi-consistent). An ETC matrix is considered consistent when the following is true: if a machine m_i executes a task j faster than machine m_j, then m_i executes all tasks faster than m_j. Inconsistency means that a machine is faster for some tasks and slower for some others. An ETC matrix is considered semi-consistent if it contains a consistent sub-matrix.
- yy indicates the heterogeneity of the tasks (hi means high, and lo means low).
- zz indicates the heterogeneity of the resources (hi means high, and lo means low).
- k numbers the instances of the same type.

We report computational results for the following 12 instances, for which we provide their Blazewicz [18] notation:

- $u\_c\_hihi.0$: $Q16|26.48 \leq p_j \leq 2892648.25|C_{max}$;
- $u\_c\_hilo.0$: $Q16|10.01 \leq p_j \leq 29316.04|C_{max}$;
- $u\_c\_lohi.0$: $Q16|12.59 \leq p_j \leq 99633.62|C_{max}$;
- $u\_c\_lolo.0$: $Q16|1.44 \leq p_j \leq 975.30|C_{max}$;
- $u\_i\_hihi.0$: $R16|75.44 \leq p_j \leq 2968769.25|C_{max}$;
- $u\_i\_hilo.0$: $R16|16.00 \leq p_j \leq 29914.19|C_{max}$;
- $u\_i\_lohi.0$: $R16|13.21 \leq p_j \leq 98323.66|C_{max}$;
- $u\_i\_lolo.0$: $R16|1.03 \leq p_j \leq 973.09|C_{max}$;
- $u\_s\_hihi.0$: $R16|185.37 \leq p_j \leq 2980246.00|C_{max}$;
- $u\_s\_hilo.0$: $R16|5.63 \leq p_j \leq 29346.51|C_{max}$;
- $u\_s\_lohi.0$: $R16|4.02 \leq p_j \leq 98586.44|C_{max}$;
- $u\_s\_lolo.0$: $R16|1.69 \leq p_j \leq 969.27|C_{max}$.

4.2. Results

We now present and discuss the results of our computational experiments. The discussion includes a comparison with other algorithms in the literature.

We first study the speedup of the algorithm as an indication of its scalability: how performance improves with the number of threads. This study helps tune the optimal number of threads for the experiments. Speedup is usually defined as:

$$S(n) = time(1)/time(n) \ , \tag{4}$$

where n is the number of machines, or processors. In the problem studied here, time is fixed to 90 seconds. We therefore replace time with the total number of evaluations.

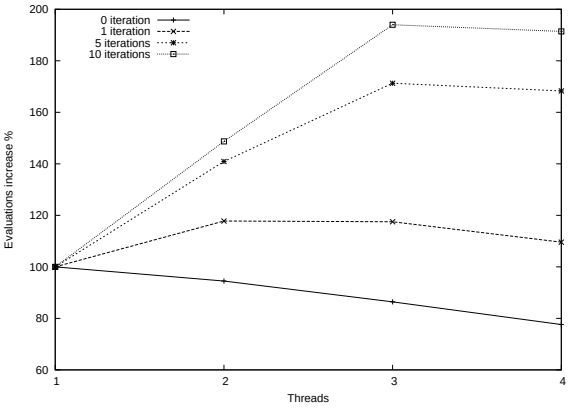

Figure 4: Speedup of the algorithm.

With time, a performance improvement corresponds to a smaller execution time, but with evaluations, an improvement corresponds to more evaluations. This leads to the following definition of speedup used in this paper:

$$S(n) = \#evaluations(n)/\#evaluations(1) \ , \quad (5)$$

where $\#evaluations(n)$ is the mean number of evaluations over 100 independent runs, and n is the number of threads.

Figure 4 shows how performance evolves with the:

- number of threads,
- number of local search iterations.

We first observe that without local search (0 iteration) the performance decreases with the number of threads. This result is essentially due to thread synchronization. Without local search, the evolution of an individual requires less computation, but the same amount of synchronization (for recombination and replacement). So the proportion of computation under synchronization increases. In addition, increasing the number of threads reduces eace. Given the partition of the population, a smaller block means that more individuals are on the boundary of the block, where the neighborhood therefore crosses block boundaries and may cause synchronization delays. The combination of these factors lead to more synchronization delays, which decreases performance when the number of threads increases.

As the number of local search iterations increases, more computation outside synchronization is performed (local search is performed on the offspring). This reduces synchronization delays, and we achieve positive speedups. Yet, we notice that with 5 or 10 local search iterations, there is no more performance gained when increasing the number of threads from 3 to 4. This is caused by the smaller block sizes. Although more time is spent in local search, the proportion of individuals on the boundary of their block increases, and with fewer individuals to process, synchronization is more frequent. Finally, the processor level 2 cache is shared across all running threads. Increasing the number of threads with little data locality negatively impacts performance. From the speedup results, we notice that 3 threads reach the maximum number of evaluations, so we adopt this model for the next studies in this paper.

Table 2: Comparison versus other algorithms in the literature. Mean makespan values.

instance	Struggle GA [19]	cMA + LTH [20]	PA-CGA 10 sec	PA-CGA
u_c_hihi.0	7752349.4	7554119.4	**7518600.7**	**7437591.3**
u_c_hilo.0	155571.48	**154057.6**	154963.6	154392.8
u_c_lohi.0	250550.9	247421.3	**245012.9**	**242061.8**
u_c_lolo.0	5240.1	**5184.8**	5261.4	5247.9
u_s_hihi.0	4371324.5	4337494.6	**4277497.3**	**4229018.4**
u_s_hilo.0	983334.6	97426.2	97841.6	**97424.8**
u_s_lohi.0	127762.5	128216.1	**126397.9**	**125579.3**
u_s_lolo.0	3539.4	**3488.3**	3535.0	3525.6
u_i_hihi.0	3080025.8	3054137.7	**3030250.8**	**3011581.3**
u_i_hilo.0	76307.9	75005.5	**74752.8**	**74476.8**
u_i_lohi.0	107294.2	106158.7	**104987.8**	**104490.1**
u_i_lolo.0	2610.2	**2597.0**	2605.5	2602.5

Next, we examine the impact of the recombination operators (opx and tpx), and the number of local search iterations (5 and 10). Figure 5 presents these results. They are obtained over 100 independent runs. Three threads are used. A box plot is provided for each instance file. In these plots, when the notches in the boxes does not overlap, we can conclude, with 95% confidence, that the true medians differ. We notice that overall, the tpx recombination operator provides better mean makespan results than opx. Furthermore, 10 iterations of our local search $H2LL$ achieve a better mean makespan than 5. With statistical significance, we can state that $tpx/10$ performs better than $opx/5$ for all instances. It finds the best mean makespans in most instances, but not in all. For the consistent instances, opx and tpx find similar mean makespan values. For the next studies in this paper, we use the tpx recombination and 10 local search iterations.

Table 2 presents a comparison of our results with others found in the literature. Results for cMA + LTH (a CGA hybridized with Tabu search) [20] and struggle GA (a non-decentralized population GA) [19] are averages over 10 independent runs, and they were taken from the original papers. We propose 2 sets of results for our algorithm, PA-CGA. One for runs of 10 seconds, another for runs of 90 seconds. For both run times, the results presented are averages of 100 independent runs. Makespan values in bold indicate the best results for an instance, or if 10 seconds of runtime of our algorithm achieves better results than the literature.

We present results for runs of 10 seconds because of the difference in computing platforms used in [20] and in our experiments. In [20], all experiments were conducted on a AMD K6 450 Mhz processor. This machine is slower than the one we use. To account for this difference, we wish to reduce the runtime in our experiments, in the same proportion. We therefore benchmark both machines, compute the performance ratio of the 2 machines, and apply

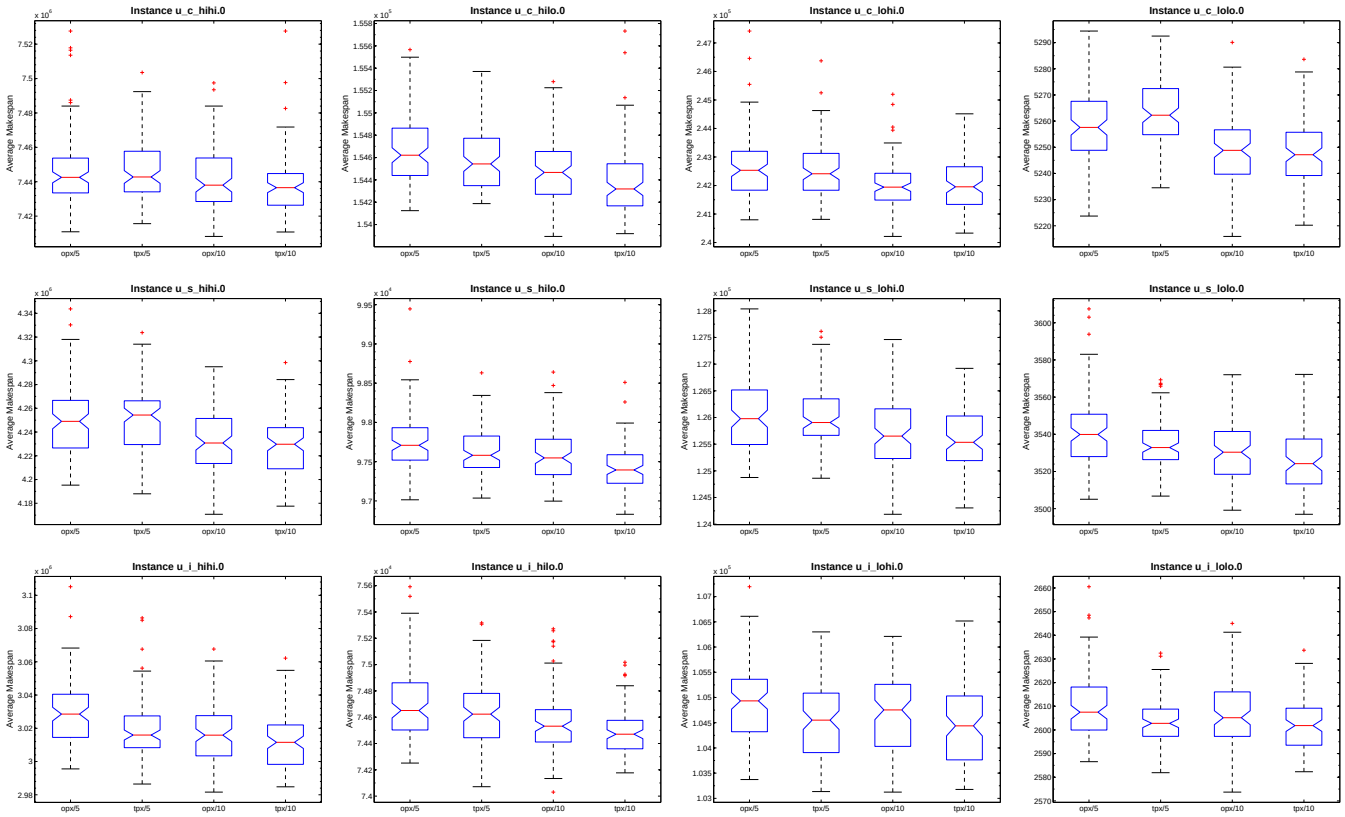

Figure 5: Comparison of recombination operators and local search iterations.

it to our runtime. Unfortunately, we do not have access to a AMD K6 450 Mhz machine. However, there exists one benchmark whose results for this machine have been published, and is available for execution on our machine. It is the program TSCP 1.7.3 [21]. One advantage of this benchmark is that it implements a combinatorial algorithm, and does not test a specific processor feature. Executing the benchmark shows a performance ratio of 9 between the 2 machines. Therefore, we provide results for runs of $90/9$ seconds, as a comparison point.

Table 2 shows that PA-CGA improves most previous results. Particularly, it provides the best results for inconsistent instances (where the performance of a machine varies from one task to another) and for instances of high heterogeneity in tasks and resources. It improves half of the results for consistent, and semi-consistent instances. It does not improve results for instances where the tasks and resources have a low heterogeneity (homogeneous). These results are useful because inconsistent instances, and instances with high task and resource heterogeneity, represent the more complex problem formulation of independent task scheduling. Also, scheduling independent near-homogeneous tasks on near-homogeneous machines can be effectively addressed with alternative simpler and faster methods, such as heuristics [9].

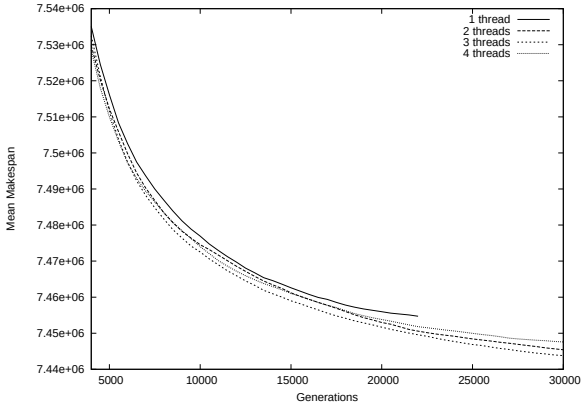

Figure 6: Evolution of the algorithm.

We can also notice that our algorithm improves the results for instances with greater makespan values.

Figure 6 shows how makespan, averaged across the population (all threads) and over 100 independent runs, evolves with the number of generations. All runs process the $u\_c\_hihi.0$ instance file. The stop condition is 90 seconds wall time. Each line corresponds to a different number of threads. In order to display differences, a subset of the domain (generations) is plotted. First, we notice that running the algorithm with 1 thread evolves for less generations than

with more threads, in the allocated time. Also, 1 thread finds worse average makespan, at any generation. It is important to note that our algorithm configured for 1 thread represents the canonical asynchronous CGA of Section 3.1. With 4 threads, we observe that the algorithm converges faster initially, but fails to reach the best solutions. Running the algorithm with 3 threads finds the best solutions.

5. Conclusion and Future Work

We presented in this paper a new parallel asynchronous CGA algorithm for multi-core processors. This algorithm was applied to the problem of independent task scheduling on a grid. We evaluated the performance of this algorithm on benchmark instances, and improved previous results. Future work will focus on increasing the parallelism of the algorithm. This means both providing greater parallelism and improving its ability to find good solutions. Since the evaluation will require machines with many cores, we will target GPU processors. Also, we will apply future parallel models on bigger benchmark instances of the independent task scheduling on grids problem.

References

[1] E. Alba and B. Dorronsoro, *Cellular Genetic Algorithms*, ser. Operations Research/Compuer Science Interfaces. Springer-Verlag Heidelberg, 2008.

[2] G. Robertson, "Parallel implementation of genetic algorithms in a classifier system," in *Proc. of the Second International Conference on Genetic Algorithms (ICGA)*, J. J. Grefenstette, Ed. L. Erlbaum Associates Inc., 1987, pp. 140–147.

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

[4] G. Luque, E. Alba, and B. Dorronsoro, *Optimization Techniques for Solving Complex Problems*. Wiley, 2009, ch. Analyzing Parallel Cellular Genetic Algorithms, pp. 49–62.

[5] ——, "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.

[6] I. Foster and C. Kesselman, *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[7] F. Pinel, B. Dorronsoro, and P. Bouvry, "A new parallel asynchronous cellular genetic algorithm for de novo genomic sequencing," in *Proceedings of the IEEE International Conference on Soft Computing and Pattern Recognition (SOCPAR09)*, 2009, pp. 178–183.

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

[9] 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. Hengsen, and R. F. 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*, vol. 61, no. 6, pp. 810–837, 2001.

[10] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *IEEE Comput.*, vol. 26, no. 6, pp. 78–86, 1993.

[11] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 42–51, 1998.

[12] D. Whitley, "Cellular genetic algorithms," in *Fifth International Conference on Genetic Algorithms (ICGA)*, S. Forrest, Ed. California, CA, USA: Morgan Kaufmann, 1993, p. 658.

[13] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, October 2002.

[14] E. Alba, B. Dorronsoro, M. Giacobini, and M. Tomassini, *Handbook of Bioinspired Algorithms and Applications*. CRC Press, 2006, ch. Decentralized Cellular Evolutionary Algorithms, pp. 103–120.

[15] IEEE and The Open Group, "Posix (ieee std 1003.1-2008, open group base specifications issue 7)," http://www.unix.org, 2008.

[16] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[17] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing task and machine heterogeneities for heterogeneous," *Journal of Science and Engineering, Special 50 th Anniversary Issue*, vol. 3, pp. 195–207, 2000.

[18] J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Scheduling subject to resource constraints: classification and complexity," *Discrete Applied Mathematics*, vol. 5, pp. 11–24, 1983.

[19] F. Xhafa, "An experimental study on GA replacement operators for scheduling on grids," in *The 2nd International Conference on Bioinspired Optimization Methods and their Applications (BIOMA)*, Ljubljana, Slovenia, October 2006, pp. 212–130.

[20] F. Xhafa, E. Alba, B. Dorronsoro, and B. Duran, "Efficient batch job scheduling in grids using cellular memetic algorithms," *Journal of Mathematical Modelling and Algorithms*, vol. 7, pp. 217–236, 2008.

[21] T. Kerrigan, "Tom kerrigan's simple chess program," http://www.tckerrigan.com/Chess/TSCP/.