10

Efficient Batch Job Scheduling in Grids Using Cellular Memetic Algorithms

Fatos Xhafa[1], Enrique Alba[2], Bernabé Dorronsoro[3], Bernat Duran[1], and Ajith Abraham[3]

[1] Dept. de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
C/Jordi Girona 1-3, 08034 Barcelona, Spain
fatos@lsi.upc.edu

[2] Dpto. de Lenguajes y Ciencias de la Computación
E.T.S.I. Informática, Campus de Teatinos
29071 Málaga, Spain
eat@lcc.uma.es

[3] Faculty of Science, Technology and Communication University of Luxembourg
6, rue Richard Coudenhove-Kalergi L-1359 Luxembourg
bernabe.dorronsoro@uni.lu

[4] Centre for Quantifiable Quality of Service in Communication Systems, Norwegian University of Science and Technology, NO-7491 Trondheim, Norway
ajith.abraham@ieee.org
http://www.softcomputing.net

Summary. Due to the complex nature of Grid systems, the design of efficient Grid schedulers is challenging since such schedulers have to be able to optimize many conflicting criteria in very short periods of time. This problem has been tackled in the literature by several different meta-heuristics, and our main focus in this work is to develop a new highly competitive technique with respect to the existing ones. For that, we exploit the capabilities of Cellular Memetic Algorithms, a kind of Memetic Algorithm with structured population, for obtaining efficient batch schedulers for Grid systems, and the resulting scheduler is experimentally tested through a Grid simulator.

Keywords: Cellular Memetic Algorithms, Job Scheduling, Grid Computing, ETC model, Makespan, Dynamic computing environment, Simulation.

10.1 Introduction

One of the main motivations of the Grid computing paradigm has been the computational need for solving many complex problems from science, engineering, and business such as hard combinatorial optimization problems, protein folding, financial modelling, etc. [19,21,22]. One key issue in Computational Grids is the allocation of jobs (applications) to Grid resources. The resource allocation problem is known to be computationally hard as it is a generalization of the standard scheduling problem. Some of the features of the Computational Grids that make

F. Xhafa, A. Abraham (Eds.): Meta. for Sched. in Distri. Comp. Envi., SCI 146, pp. 273–299, 2008.
springerlink.com

the problem challenging are the high degree of heterogeneity of resources, their connection with heterogenous networks, the high degree of dynamics, the large scale of the problem regarding number of jobs and resources, and other features related to existing local schedulers, policies on resources, etc. (see Chapter 1, this volume).

Meta-heuristic approaches have shown their effectiveness for a wide variety of hard combinatorial problems and also for multi-objective optimization problems. In this work we address the use of Cellular Memetic Algorithms (cMAs) [3, 4, 5, 6, 16] for efficiently scheduling jobs to Grid resources. cMAs are population-based algorithms that maintain a structured population as opposed to GAs or MAs of unstructured population. Research on cMAs has shown that, due to the structured population, this family of algorithms is able to better control the tradeoff between the exploitation and exploration of the solution space with respect to other non-structured algorithms [3, 4, 5]. It should be noted that this feature is very important if high quality solutions are to be found in a very short time. This is precisely the case of the job scheduling in Computational Grids whose highly dynamic nature makes indispensable the use of schedulers that would be able to deliver high quality planning of jobs to resources very fast in order to deal with the changes of the Grid. On the other hand, population-based heuristics are potentially good also for solving complex problems in the long run providing, for many problems, near optimal solutions. This is another interesting feature to explore regarding the use of cMAs for the job scheduling problem. The evidence reported in the literature that cMAs are capable to maintain a high diversity of the population in many generations suggests that cMAs could be appropriate for scheduling jobs that periodically arrive in the Grid system since in this case the Grid scheduler would dispose longer intervals of time to compute the planning of jobs to Grid resources. Finally, cMAs are used here to solve the bi-objective case of the job scheduling, namely makespan and flowtime are simultaneously optimized.

Many different cMA configurations have been developed and compared in this study on a benchmark of static instances of the problem (proposed by Braun et al. [9]). After that, we have also studied the behavior of the best obtained configuration in a more realistic benchmark of dynamic instances. Our algorithms will be validated by comparing the obtained results versus other results in the literature for the same studied benchmarks (both the static and the dynamic ones). Moreover, we studied the robustness of our cMA implementation since robustness is a desired property of Grid schedulers, which are very changing in nature. Because the cMA scheduler is able to deliver very high quality planning of jobs to Grid nodes, it can be used to design efficient dynamic schedulers for real Grid systems. Such dynamic schedulers are obtained by running the cMA-based scheduler in batch mode for a very short time to schedule jobs arriving in the systems since the last activation of the cMA scheduler.

This chapter is organized as follows. We give in Section 10.2 a description of the job scheduling in computational grids. The cMAs and their particularization for job scheduling in Grids together with the tuning process for the values of

the parameters of the algorithm are given in Section 10.3. Some computational results as well as their evaluation for a benchmark of static instances are presented in Section 10.4. In Section 10.5, the best of the tested cMA configurations are evaluated in the more realistic case of dynamic instances, and the results are compared versus those of other algorithms found in the literature. Finally, we end in Section 10.7 with some conclusions.

10.2 The Batch Job Scheduling on Grids

In this work we consider the version of the problem[1] that arises quite frequently in parameter sweep applications, such as Monte-Carlo simulations [11]. In these applications, many jobs with almost no interdependencies are generated and submitted to the Grid system. In fact, more generally, the scenario in which the submission of independent jobs to a Grid system is quite natural given that Grid users independently submit their jobs or applications to the Grid system and expect an efficient allocation of their jobs/applications. We notice that the efficiency means that we are interested to allocate jobs as fast as possible and to optimize two conflicting criteria: *makespan* and *flowtime*.

In our scenario, jobs are originated from different users/applications, have to be completed in unique resource unless it drops from the Grid due to its dynamic environment (*non-preemptive* mode), are independent of each other and could have their hardware and/or software requirements over resources. On the other hand, resources could dynamically be added/dropped from the Grid, can process one job at a time, and have their own computing characteristics regarding 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, 15, 18]), 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) *jobs* to be scheduled.
- A *number* of heterogeneous *machines* candidates to participate in the planning.
- The *workload* of each job (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 jobs.
- The Expected Time to Compute (ETC) matrix ($nb\_jobs \times nb\_machines$) in which $ETC[i][j]$ is the expected execution time of job i in machine j.

10.2.1 Optimization Criteria

We consider the job scheduling as a bi-objective optimization problem, in which both makespan and flowtime are simultaneously minimized. These criteria are defined as follows:

[1] The problem description and simultaneous optimization criteria are given in Chapter 1 and are reproduced here for completeness.

- *Makespan* (the finishing time of latest job) defined as $\min_S \max\{F_j : j \in Jobs\}$,
- *Flowtime* (the sum of finishing times of jobs), that is, $\min_S \sum_{j \in Jobs} F_j$,

where F_j is the finishing time of job j in schedule S.

For a given schedule, it is quite useful to define the *completion time* of a machine, which indicates the time in which the machine will finalize the processing of the previous assigned jobs as well as of those already planned for the machine. Formally, for a machine m and a schedule S, the completion time of m is defined as follows:

$$completion[m] = ready_m + \sum_{j \in S^{-1}(m)} ETC[j][m] \ . \tag{10.1}$$

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

$$\min_S \max\{completion[i] \mid i \in Machines'\} \ . \tag{10.2}$$

In order to deal with the simultaneous optimization of the two objectives we have used a simple weighted sum function of makespan and flowtime, which is possible since both parameters are measured in the same unit (time units). This way of tackling multiobjective optimization problems is widely accepted in the literature [12, 13], and its drawbacks are well known: only a single solution from the Pareto front (a set containing the best non-dominated solutions to the problem) is found in each run, and only solutions located in the convex region of the Pareto front will be found. However, the use of a weighted function is justified in our case by the convex search space of the considered problem and also by the need of providing a unique solution to the grid system, since there is not any decision maker to select the most suitable solution from a set of non-dominated ones.

The makespan and flowtime values are in incomparable ranges, since flowtime has a higher magnitude order over makespan, and its difference increases with the number of jobs and machines to be considered. For this reason, the value of mean flowtime, $flowtime/nb\_machines$, is used instead of flowtime. Additionally, both values are weighted in order to balance their importance. Fitness value is thus calculated as:

$$fitness = \lambda \cdot makespan + (1 - \lambda) \cdot mean\_flowtime \ , \tag{10.3}$$

where λ has been *a priori* fixed after a preliminary tuning process to the value $\lambda = 0.75$ for the studies made in this work. Hence, we are considering in this work the makespan as the most important objective to optimize, while we give less importance to the total flowtime obtained in our solutions.

10.3 A cMA for Resource Allocation in Grid Systems

We present in this section a description of the cMA we are proposing in this work (Section 10.3.1) and its application to the batch job scheduling problem (Section 10.3.2).

10.3.1 Cellular Memetic Algorithms

In Memetic Algorithms (MAs) the population of individuals could be unstructured or structured. In the former, there is no relation between the individuals of the population while in the latter individuals can be related to only some other specific individuals of the population. The structured MAs are usually classified into *coarse-grained* model and *fine-grained* (Cellular MAs) model [4, 5, 6, 16]. In Cellular MAs the individuals of the population are spatially distributed forming neighborhoods and the evolutionary operators are applied to neighbor individuals making thus cMAs a new family of evolutionary algorithms. As in the case of other evolutionary algorithms, cMAs are high level algorithms whose description is independent of the problem being solved. Thus, for the purposes of this work, we have considered the cMA template given in Algorithm 10.1.

As it can be seen, this template is quite different from the canonical cGA approximation [4, 5], in which individuals are updated in a given order by applying the recombination operator to the two parents and the mutation operator to the obtained offspring. In the case of the proposed algorithm in this work, mutation and recombination operators are applied to individuals independently of each other, and in different orders. This model was adopted after a previous experimentation, in which it performed better than the cMA following the canonical model for the studied problems. After each recombination (or mutation), a local search step is applied to the newly obtained solution, which is then evaluated. If this new solution is better than the current one, it replaces the latter in the population. This process is repeated until a termination condition is met.

10.3.2 Application of the cMA to job Scheduling

Given the generic template showed in Algorithm 10.1, we proceed in this section to define the different parameters and operators we will use for solving the problem of batch job scheduling in grids. In order to efficiently solve the problem, we have to particularize the template with operators incorporating some specific knowledge of the problem at hand. The objective is to design an efficient algorithm for optimizing the QoS and productivity of grid systems. For that, we will use genetic operators focussed in balancing the load of all the available machines, and taking into account the presence of heterogeneous computers. We give next the description of the cMA particularization for job scheduling.

Regarding the problem representation, a feasible solution, *schedule*, is considered as a vector of size the number of jobs (nb_jobs) in which its jth position (an integer value) indicates the machine where job j is assigned: $schedule[j] = m, m \in \{1, \ldots, nb\_machines\}$.

Algorithm 10.1. A Cellular MA template

```
Initialize the mesh of n individuals P(t=0);
Initialize permutations rec_order and mut_order;
For each i ∈ P, LocalSearch(i);
Evaluate(P);
while not stopping condition do
  for j = 1...#recombinations do
    SelectToRecombine S ⊆ N_P[rec_order.current];
    i' = Recombine(S);
    LocalSearch(i'); Evaluate(i');
    Replace P[rec_order.current] by i';
    rec_order.next();
  end for
  for j = 1...#mutations do
    i = P[mut_order.current()];
    i' = Mutate(i);
    LocalSearch(i'); Evaluate(i');
    Replace P[rec_order.current] by i';
    rec_order.next();
  end for
  Update rec_order and mut_order;
end while
```

As it can be seen in Algorithm 10.1, many parameters are involved in the cMA template. Tuning these parameters is a crucial step in order to achieve a good performance, since they influence in a straightforward way on the search process. The tuning process was done by using randomly generated instances of the problem according to the ETC matrix model. This way we would expect a robust performance of our cMA implementation since no specific instance knowledge is used in fixing the values of the parameters. An extensive experimental study was done in order to identify the best configuration for the cMA. Thus, we experimentally studied the choice of the local search method, the neighborhood pattern, the selection, recombination and mutation operators, and the cell update orders. The tuning process was made step by step, starting from an initial configuration set by hand, and adding in each step the tuned parameters of the previous ones. We give in Figs. 10.2 to 10.8 the graphical representation for the makespan reduction of the cMA with the considered parameters. The results are obtained after making 20 independent runs in standard configuration computer.

Population's topology and neighborhood structure

Both the topology of the population and the neighborhood pattern are very important parameters in deciding the selective pressure of the algorithm and, therefore, they have a direct influence on the tradeoff between exploration and exploitation of the algorithm [2, 7]. The topology of the population is a two-dimensional toroidal grid of pop_height × pop_width size. Regarding the neighborhood patterns, several well-known patterns are used for this work: L5 (5 individuals), L9 (9 individuals), C9 (9 individuals) and C13 (13 individuals) (see Fig. 10.1). Additionally, in our quest for efficiency, we have considered the case in which the neighborhood is equal to the whole population, so an individual can interact with any other one in the population. Using this boundary

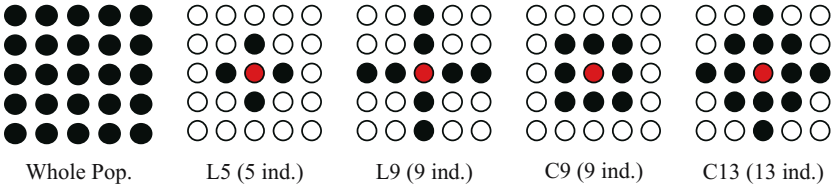

Fig. 10.1. Neighborhood patterns

neighborhood we remove a typical feature of cellular populations from our cMA, namely, the isolation by distance. The pursued effect is to accelerate the convergence of the algorithm up to the limit in order to check if it is profitable for the cMA.

We study in Fig. 10.2 the effects of using the different neighborhood structures previously proposed in our cMA in order to identify the pattern that leads to the best performance for the job scheduling problem. As it can be seen, we obtain from this study that the obtained makespan worsens when increasing the radius of the neighborhood (refer to [7] for a definition of the neighborhood radius). Among the tested neighborhoods, L5 and C9 (those with the smallest radii) perform the best exploration/exploitation tradeoffs of the algorithm for this problem. Between them, we can see that L5 yields a very fast reduction, although C9 performs better in the "long run" (see Fig. 10.2).

Finally, the case of considering the whole population as the neighborhood throws the worst performance (slowest convergence) of the algorithm. This is probably because the diversity in the population is quickly lost and thus the speed of the population evolution becomes very slow.

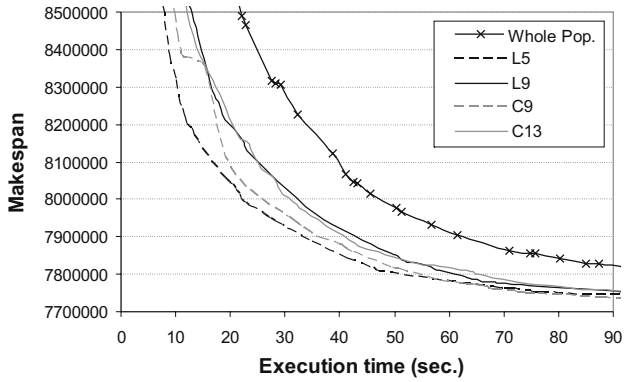

Fig. 10.2. Makespan reduction obtained with different neighborhood patterns (Makespan values are in arbitrary time units)

Population initialization

In this work we make use of some problem knowledge for creating the initial population of individuals. This way, one individual is generated using the *Longest Job to Fastest Resource - Shortest Job to Fastest Resource (LJFR-SJFR)* heuristic [1], while the rest are randomly obtained from the first individual by large perturbations. The LJFR-SJFR method has been chosen because it tries to simultaneously minimize both makespan and flowtime. LJFR minimizes the makespan and it is alternated with the SJFR which minimizes the flowtime. The method starts by increasingly sorting jobs with respect to their workload. At the beginning, the first *nb_machines* longest jobs are assigned to the *nb_machines* idle machines (the longest job to the fastest machine and so on). For the remaining jobs, at each step the fastest machine (that has finished its jobs) is chosen to which is assigned alternatively either the shortest job (SJFR) or the longest job (LJFR).

Cell updating

Unlike many unstructured MAs, in cMAs the population is kept constant by applying cell updating mechanisms by which an individual of the population is updated with a new offspring obtained by either recombination or mutation process (see later for the definition of these two operators). Two well-known methods of cell updating are the synchronous and asynchronous updating. For the purpose of this work, we have considered the asynchronous updating since it is less computationally expensive and usually shows a good performance in a very short time [8], which is interesting for the scheduling problem given the dynamic nature of Grid systems. In the asynchronous mode, cell updating is done sequentially (an individual is aware of other neighbor individual updates during the same iteration). The following asynchronous mechanisms have been implemented and experimentally studied for our job scheduling problem:

- **Fixed Line Sweep (FLS):** The individuals of the *grid* are updated in a sequential order row by row.
- **Fixed Random Sweep (FRS):** The sequence of cell updates is at random. This sequence is defined at the beginning of the algorithm and it is the same during all the cMA iterations.
- **New Random Sweep (NRS):** At each iteration, a new cell update sequence (at random) is applied.

It should be noted that recombination and mutation are independent processes in our cMAs (cf. `rec_order` and `mut_order` in the cMAs template) and therefore different update orders are used for them. Next, we study some different update policies and probabilities for applying them for the recombination and mutation steps.

In Fig. 10.3 we provide a study of the three proposed update policies for the recombination operator applied with two different probabilities. As regards to the cell updating for the recombination operator, the three considered mechanisms

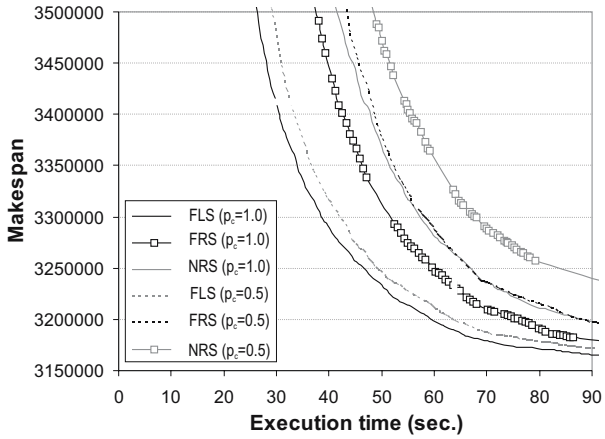

Fig. 10.3. Makespan reduction with different recombination orders and probabilities (p_c)

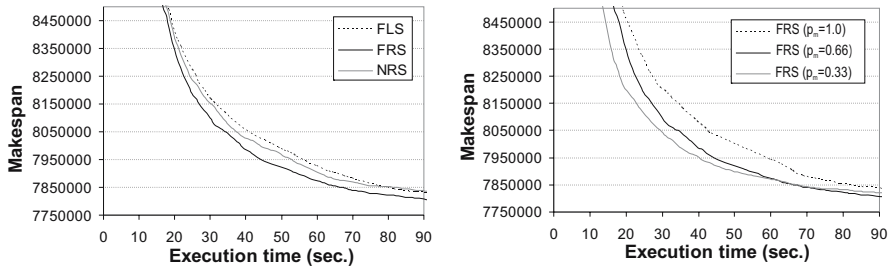

Fig. 10.4. Makespan reduction obtained with different mutation orders –left– and probabilities (p_m) –right

performed similarly, the FLS being the best performer (see Fig. 10.3). For the three update policies, the case of always recombining the individuals ($p_c = 1.0$) is advantageous versus applying the operator with probability $p_c = 0.5$.

Regarding the mutation operator, it can be seen in Fig. 10.4 (left hand plot) that, like in the case of the recombination operator, the three update policies perform in a similar way, being FRS slightly better than the other two ones. In the right hand plot of this same figure we study three different probabilities of applying the mutation operator when using FRS. The main result that can be drawn from this study is that the two lower probabilities ($p_m = 0.66$ and $p_m = 0.33$) perform better than the highest one ($p_m = 1.0$). When comparing these two lowest probabilities between them, one can notice that the case $p_m = 0.33$ converges faster, but after the 90 seconds allowed for the execution using $p_m = 0.66$ seems to be beneficial.

Selection operator for recombination

We have considered in this work the use of six different well-known selection policies in our cMA, namely linear ranking (LR), N-tournament (Ntour) with $N = 2, 3, 5,$ and 7, and selecting the best individual in the neighborhood (Best). The results of our experiments are given in Fig. 10.5. As it can be seen, the slowest convergence is given by both linear ranking and binary tournament (Ntour with $N = 2$), although at the end of the run the makespan found using these two selection methods is close to that of the other compared ones, for which the convergence is faster at the beginning of the run, although its speed is drastically slowed after a few seconds. From all the compared selection methods, the one reporting the best makespan at the end of the run is Ntour with $N = 3$.

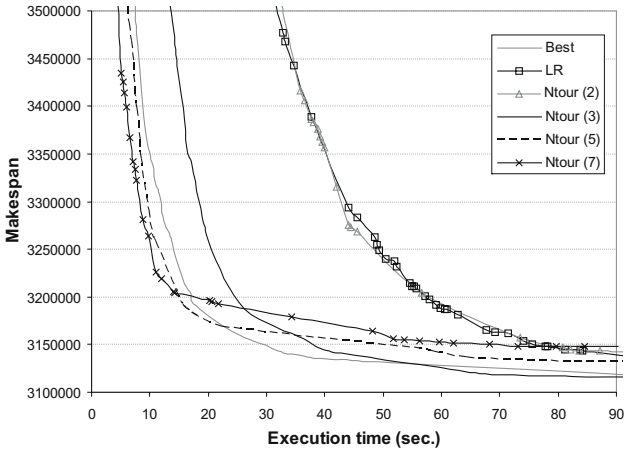

Fig. 10.5. Makespan reduction obtained with different selection methods

Recombination operator

Three recombination operators, very well known in the literature, were tested in this study for tuning our cMA. They are the *one-point* (OP2), the *uniform* (Uni2), and the *fitness-based* (FB2) recombination. The *one-point* operator lies in splitting the two chromosomes into two parts (in a randomly selected point), and joining each part of one parent chromosome with the other part of the chromosome of the second parent. In the case of the *uniform* recombination, an offspring is constructed by selecting for each gene the value of the corresponding gene of one of the two parents with equal probability. Finally, in the case of the *fitness-based* recombination both the structure and the relative fitness of the two parents are taken into account. The offspring is obtained as follows. Let us suppose that the two parents are P_1 and P_2, being $P_1[i]$ the i^{th} gene of P_1 and f_{P_1} its fitness. The offspring is noted as C. If the two parents have the same value for a given gene i ($P_1[i] = P_2[i]$) this value is adopted for the same gene

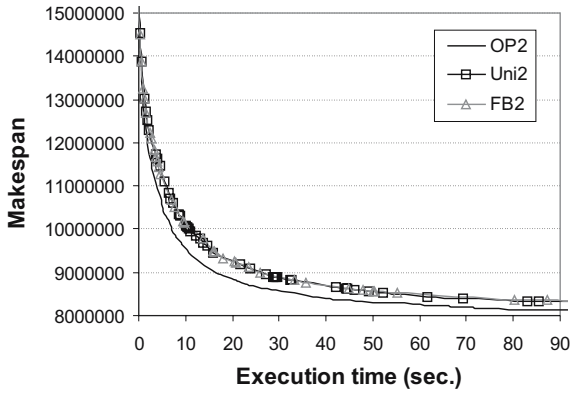

Fig. 10.6. Makespan reduction obtained with different recombination operators

of the offspring $C[i] = P_1[i]$. In other case, when $P_1[i] \neq P_2[i]$, $C[i] = P_1[i]$ with probability $p = f_{P_2}/(f_{P_1} + f_{P_2})$, while $C[i] = P_2[i]$ with probability $1 - p$.

From the results showed in Fig. 10.6, the *one-point* method has been chosen as the one reporting the best performance from the three compared recombination operators. The other two tested recombination operators (Uni2 and FB2) perform in a similar way, and slightly worse than OP2.

Mutation operator

We have tested four different mutation operators in our cMA. They are *move*, *swap*, *both*, and *rebalance*:

- *Move* is a simple operator that lies in changing the location of a given job in the chromosome of the individual, i.e. it assigns the machine of job i to job j.
- *Swap* exchanges the value of two genes. In our problem, this means that we are exchanging the machines assigned to two different jobs.
- *Both*. In this case we are applying one of the two previously explained operators (*move* and *swap*) with equal probability.
- *Rebalance*. The mutation is done by *rebalancing* of machine loads of the given schedule. The load factor of a machine m is defined as load_factor(m) = completion[m]/makespan (load_factor(m) $\in (0, 1]$). The idea behind this choice is that in a schedule, some machines could be overloaded (when its completion time is equal to the current makespan –load_factor(m) = 1–) and some others less overloaded (regarding the overloaded machines, we sort the machines in increasing order of their completion times and 25% first machines are considered less overloaded), in terms of their completion times. It is useful then to mutate the schedule by a load balancing mechanism, which transfers a job assigned to an overloaded machine to another less loaded machine.

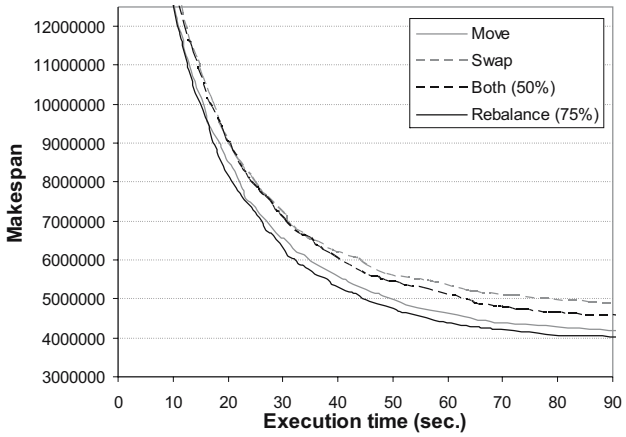

Fig. 10.7. Makespan reduction obtained with different mutation operators

Three of the studied mutation operators (*move*, *swap*, and *both*) are generic ones, while *rebalance* was specifically designed for this problem. They all are compared in Fig. 10.7. As it can be seen, the best performance is given by the *rebalance* operator (the unique specific method of the studied ones). Comparing the generic operators, *swap* is the worst one, and *move* is the best, being the results obtained by *both* between they two.

Local search methods

Local search is a proper feature of Memetic Algorithms. As it can be seen from the template of Algorithm 10.1, each individual is improved by a local search both after being generated by the recombination operator and after being mutated. Improvement of the descendants is thus done not only by means of genetic information but also by local improvements. The presence of this local search method in the algorithm does not increase selection pressure too much due to the exploration capabilities intrinsic to the cellular model. Four local search methods have been implemented and experimentally studied. These are the Local Move (LM), Steepest Local Move (SLM), Local Minimum Completion Time Swap (LMCTS), and Local Tabu Hop (LTH).

- LM is similar to the mutation operator (a randomly chosen job is transferred to a new randomly chosen machine).
- In SLM method, the job transfer is done to the machine that yields the best improvement in terms of the reduction of the completion time.
- In LMCTS method, two jobs assigned to different machines are swapped; the pair of jobs that yields the best reduction in the completion time is applied.
- Local Tabu Hop is a local search method based on the Tabu Search (TS) meta-heuristic. The main feature of TS [17] is that it maintains an

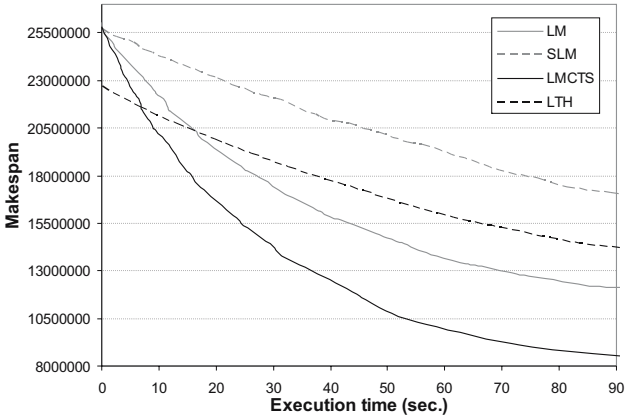

Fig. 10.8. Makespan reduction obtained with four local search methods

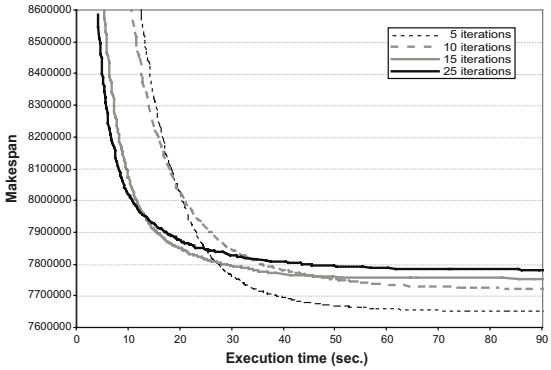

Fig. 10.9. Makespan reduction obtained with different intensities of the local search method

adaptive memory of forbidden (tabu) movements in order to avoid cycling among already visited solutions and thus escape from local optimal solutions. In the LTH algorithm for job scheduling, the implemented neighborhood relationship is based on the idea of the load balancing. The neighborhood of solutions consists of all those solutions to which we can reach via *swap* of the tasks of an overloaded resource with those of the less overloaded ones, or via *move* of tasks of an overloaded resource to the less overloaded resources. LTH is essentially a *phase* of Tabu Search and is taken from the Tabu Search implementation for the problem by Xhafa et al. [23].

In Fig. 10.8 we compare the behavior of our cMAs implementing the four proposed local search methods. From that graphical representation we can easily observe that the LMCTS method performs best among the four considered local

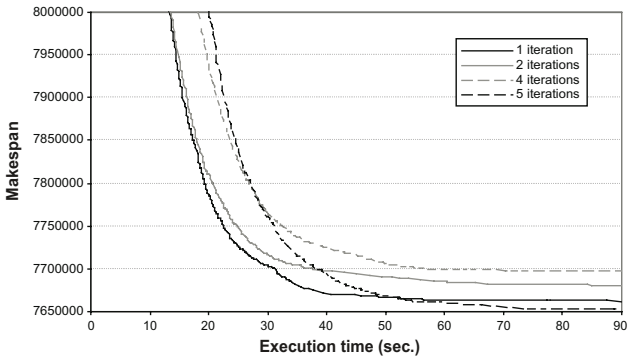

Fig. 10.10. Makespan reduction obtained with different maximum allowed iterations of the local search method when the solution is not improved

search methods. In fact, a clear difference in the behavior of the considered local search methods is observed, though all of them provide an accentuated reduction in the makespan value (see Fig. 10.8).

The bad behavior of the cMA using LTH is probably because this local search method is very heavy (computationally speaking) with respect to the other compared ones, and also the termination condition of the cMA is very hard (only 90 seconds of execution). Thus, the cMA only has time for making a few generations before the termination condition is met. Hence, it should be interesting to try some other parameters in order to reduce the number of LTH steps made by cMA+LTH in each generation, what hopefully should lead us to better results.

We present in Fig. 10.9 a study of the influence of the number of iterations of the LMCTS local search algorithm in the behavior of the cMA. Specifically, we study the cases of performing 5, 10, 15, and 25 iterations. As it can be seen in the figure, the smaller the number of iterations is the slower the convergence of the algorithm, and also the better the resulting makespan. Hence, the use of a strong local search provokes a premature convergence of the population, and this fast lost of diversity induces a bad behavior into the algorithm.

Once the number of iterations of the local search step is set, there is still one parameter to be tuned for the local search. This parameter is the number of iterations of the local search to perform even if no improvements were obtained in the previous ones. We present in Fig. 10.10 a study in which the cases of performing 1, 2, 4, and 5 iterations without any improvement are analyzed (recall that the maximum number of iterations was previously set to 5). From the results shown in Fig. 10.10 we decided to set the number of iterations of the local search to 5 even if no improvements are found.

Population Size and Shape and Replacement Policy

In this final step of our tuning procedure, we set the population size and shape as well as the replacement policy that we will use in our experiments. We compare

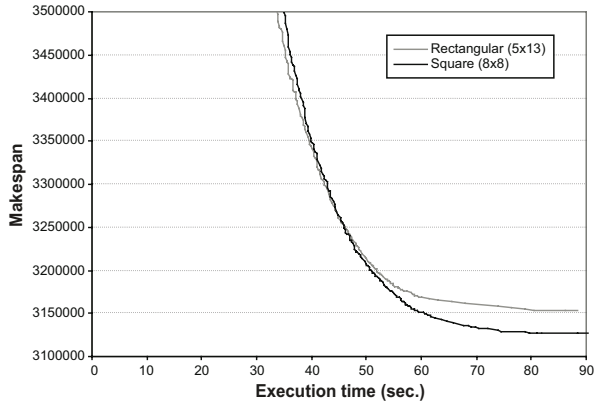

Fig. 10.11. Makespan reduction obtained with two different population shapes

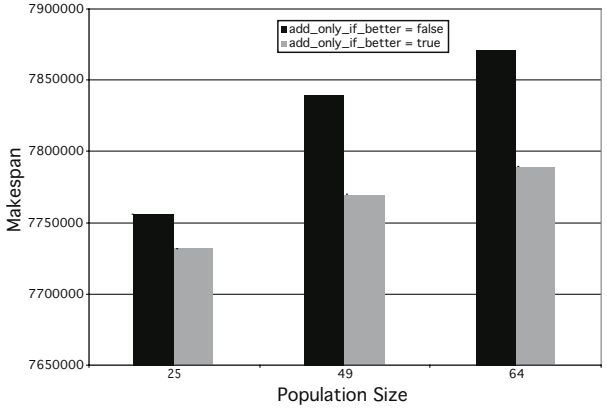

Fig. 10.12. Makespan obtained with two different replacement methods and three (square) population sizes

in Fig. 10.11 the behavior of our cMA with two populations of different shapes but having approximately the same size. The reason for this study is that the shape of the population markedly influences the behavior of the search in cellular evolutionary algorithms [3, 2]. The populations compared in Fig. 10.11 are a rectangular ones composed by 65 individuals arranged in a 5×13 mesh and a square 8×8 individuals population. As it can be seen in the figure, the latter performs better than the former for the studied instance.

We now study the influence of the replacement policy of new individuals into the population. Specifically, we considered two different options, namely, allowing that worse individuals can replace the current ones in the population ($add\_only\_if\_better = false$) or not ($add\_only\_if\_better = true$). As it can be seen in

Fig. 10.12, we always obtained better (lower) makespan values when individuals in the population can only be replaced by offsprings having better fitness values.

Additionally, we can see in Fig. 10.12 that the smallest of the three tested populations was the one providing the best makespan value (the three populations have square shape). The reason is that the use of a larger population allows to maintain the diversity for longer, but as a consequence the convergence speed is slowed down, so the algorithm generally requires a longer time to converge. This property is desirable for very difficult problems and large computation times. However, the computational time is fixed and very limited in our case of study, so it is desirable to enhance the exploitation capabilities of our algorithm.

10.4 Computational Results on Static Instances

After tuning our cMA on a set of random instances of the problem according to the ETC matrix model in Section 10.3.2, we present in this section some computational results obtained with our tuned cMAs for the benchmark of instances by Braun et al. [9] for distributed heterogenous systems. This benchmark is described in the next section, while the results of our algorithm are discussed and compared versus those obtained by other algorithms in Section 10.4.2.

10.4.1 Benchmark Description

The instances of this benchmark are classified into 12 different types of *ETC* matrices, each of them consisting of 100 instances, according to three parameters: job heterogeneity, machine heterogeneity and consistency. 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–consistent, i–inconsistent, and s means semi-consistent). An ETC matrix is considered consistent when, if a machine m_i executes job j faster than machine m_j, then m_i executes all the jobs faster than m_j. Inconsistency means that a machine is faster for some jobs 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 jobs (hi means high, and lo means low).
- zz indicates the heterogeneity of the resources (hi means high, and lo means low).

Note that all instances consist of 512 jobs and 16 machines. We report computational results for 12 instances, which are made up of three groups of four instances each. These three groups represent different Grid scenarios regarding the computing capacity. The first group corresponds to consistent *ETC* matrices (for each of them combinations between *low* and *high* are considered), the second represent instances of inconsistent computing capacity and the third one to semi-consistent computing capacity.

10.4.2 Evaluation and Discussion

In this section we present and discuss the results obtained by our algorithms, and compare them versus some other algorithms in the literature. Specifically, we propose two different cMAs: cMA+LMCTS and cMA+LTH. First, we compare the results obtained with the two proposed cMAs. Since one of these versions uses as a local search the Local Tabu Hop, we also compare the obtained results with those obtained by Tabu Search implementation by Xhafa et al. [23]. The algorithms run for 90 seconds (a single run) and 10 runs per instance are made. These decisions are the same than those adopted for the compared algorithms in order to make fair comparisons, since the compared results are directly taken from the original papers.

Table 10.1. Parameterization of cMA+LMCTS

Termination condition	Maximum of 90 seconds running
Population size	5×5
Probability of recombination	$p_c = 1.0$
Probability of mutation	$p_m = 0.5$
Population initialization	LJFR-SJFR (Longest / Shortest Job to Fastest Resource)
Neighborhood pattern	C9
Recombination order	FLS (Fixed Line Sweep)
Mutation order	NRS (New Random Sweep)
Selection method	3-Tournament
Recombination operator	One-Point recombination
Mutation operator	Rebalance
Local search method	LMCTS (Local Minimum Completion Time Swap)
Number of iterations of the local search	5
Replacement policy	Replace if better

The resulting configuration for cMA+LMCTS we decided to use after the initial tuning step made in Section 10.3.2 is given in Table 10.1. The parameterizations for cMA+LTH is similar to the one shown in Table 10.1, but in this case the population was set to 3×3 in order to reduce the number of local search steps due to the high computational requirements of LTH (see Section 10.3.2). Because of the small population used in this case, we adopt the L5 neighborhood pattern for cMA+LTH.

We give in Table 10.2 the computational results[2] for the makespan objective, where the first column indicates the name of the instance, and the other three ones present the average makespan with standard deviation (in %) obtained by the two proposed CMA algorithms (cMA+LMCTS and cMA+LTH) and TS [23]. Again, the results are averaged over 10 independent runs of the algorithms for every instance. The algorithm cMA+LMCTS provides the worst results in terms of average makespan, while the other proposed cellular memetic algorithm, cMA+LTH, is the best one for all the consistent instances, and it is the best performing algorithm if we do not take into account the inconsistent instances. This observation is interesting if the Grid characteristics were known in advance, since cMA+LTH seems to be more appropriate for consistent

[2] Values are in arbitrary time units.

Table 10.2. Comparison of the three proposed algorithms. Average makespan values.

Instance	cMA+LMCTS		TS		cMA+LTH	
u_c_hihi.0	7700929.751	±0.73%	7690958.935	±0.28%	**7554119.350**	±0.47%
u_c_hilo.0	155334.805	±0.13%	154874.145	±0.41%	**154057.577**	±0.10%
u_c_lohi.0	251360.202	±0.62%	250534.874	±0.59%	**247421.276**	±0.47%
u_c_lolo.0	5218.18	±0.30%	5198.430	±0.52%	**5184.787**	±0.07%
u_i_hihi.0	3186664.713	±1.80%	**3010245.600**	±0.26%	3054137.654	±0.83%
u_i_hilo.0	75856.623	±0.79%	**74312.232**	±0.35%	75005.486	±0.31%
u_i_lohi.0	110620.786	±1.72%	**103247.354**	±0.42%	106158.733	±0.54%
u_i_lolo.0	2624.211	±0.83%	**2573.735**	±0.39%	2597.019	±0.39%
u_s_hihi.0	4424540.894	±0.85%	**4318465.107**	±0.28%	4337494.586	±0.71%
u_s_hilo.0	98283.742	±0.47%	**97201.014**	±0.56%	97426.208	±0.21%
u_s_lohi.0	130014.529	±1.11%	**125933.775**	±0.38%	128216.071	±0.83%
u_s_lolo.0	3522.099	±0.55%	3503.044	±1.52%	**3488.296**	±0.19%

Table 10.3. Comparison versus other algorithms in the literature. Average makespan values.

Instance	Braun et al. GA	GA (Carretero&Xhafa)	Struggle GA (Xhafa)	cMA+LTH
u_c_hihi.0	8050844.50	7700929.75	7752349.37	**7554119.35**
u_c_hilo.0	156249.20	155334.85	155571.80	**154057.58**
u_c_lohi.0	258756.77	251360.20	250550.86	**247421.28**
u_c_lolo.0	5272.25	5218.18	5240.14	**5184.79**
u_i_hihi.0	3104762.50	3186664.71	3080025.77	**3054137.65**
u_i_hilo.0	75816.13	75856.62	76307.90	**75005.49**
u_i_lohi.0	107500.72	110620.79	107294.23	**106158.73**
u_i_lolo.0	2614.39	2624.21	2610.23	**2597.02**
u_s_hihi.0	4566206.00	4424540.89	4371324.45	**4337494.59**
u_s_hilo.0	98519.40	98283.74	983334.64	**97426.21**
u_s_lohi.0	130616.53	130014.53	**127762.53**	128216.07
u_s_lolo.0	3583.44	3522.10	3539.43	**3488.30**

and semi-consistent Grid scenarios. Moreover, we consider that cMA+LTH is a more robust algorithm with respect to TS because the standard deviation values of the results obtained by the former are lower than those of the latter, in general.

We believe that it is possible to improve the results of cMA+LTH if we apply longer steps of the LTH method. Additionally, as it happened in [4,5] for the case of the satisfiability problem, we believe that the memetic algorithm (cMA+LTH) should outperform the local search by itself (TS) for larger instances of the problem. Moreover, it makes sense in our case to solve much larger instances of the problem, since we are tackling grids composed by only 16 machines in this preliminary study, and it is desirable to solve instances including hundreds or even thousands of processors.

The comparison of our best cMA (cMA+LTH) with three other versions of GAs taken from the literature is given in Table 10.3. Like in the case of Tables 10.2 and 10.3, values are the average makespan and standard deviation obtained after 10 independent runs. The compared algorithms are the Braun et al. GA [9], the GA by Carretero and Xhafa [10], the Struggle GA [26], and our best memetic algorithm cMA+LTH. For all the compared algorithms, the termination condition is set to a 90 seconds runtime. As it can be seen, cMA+LTH is

Table 10.4. Comparison versus other algorithms in the literature. Average flowtime values.

Instance	LJFR-SJFR	Struggle GA (Xhafa)	TS	cMA+LMCTS	cMA+LTH
u_c_hihi.0	2025822398.7	1039048563.0	1043010031.4	**1037049914.2**	1048630695.5
u_c_hilo.0	35565379.6	27620519.9	27634886.7	**27487998.9**	27684456.0
u_c_lohi.0	66300486.3	34566883.8	34641216.8	**34454029.4**	34812809.9
u_c_lolo.0	1175661.4	917647.31	919214.3	**913976.2**	922378.0
u_i_hihi.0	3665062510.4	379768078.0	**357818309.3**	361613627.3	370506405.1
u_i_hilo.0	41345273.2	12674329.1	**12542316.2**	12572126.6	12754803.6
u_i_lohi.0	118925453.0	13417596.7	**12441857.7**	12707611.5	12975406.6
u_i_lolo.0	1385846.2	440729.0	**437956.9**	439073.7	445529.3
u_s_hihi.0	2631459406.5	524874694.0	515743097.6	**513769399.1**	532276376.7
u_s_hilo.0	35745658.3	16372763.2	16385458.2	**16300484.9**	16628576.7
u_s_lohi.0	86390552.3	15639622.5	15255911.2	**15179363.5**	15863842.1
u_s_lolo.0	1389828.8	598332.7	597263.2	**594666.0**	605053.4

the best one of the four compared algorithms for all the studied instances, with the exception of the semi-consistent instance with low heterogeneity of jobs and high heterogeneity of the resources (`u_s_lohi.0`), for which cMA+LTH is the second best algorithm, just after the Struggle GA.

Additionally, when comparing cMA+LMCTS against the three other versions of GAs shown in Table 10.3 (Braun et al. GA, Carretero&Xhafa's GA [10] and Xhafa's Struggle GA [26]), cMA+LMCTS obtains better schedules than the compared GAs for half of the considered instances, and for the rest of the instances, the solutions found by cMA+LMCTS have a similar quality than the best of the other three GAs.

Computational results for flowtime parameter are given in Table 10.4 wherein we compare the average flowtime value obtained after 10 independent runs by the *ad hoc* heuristic LJFR-SJFR, the Xhafa's Struggle GA [26], Xhafa et al. TS [23] and the two cMAs proposed in this work. As it can be seen, the improvement made by the two cMAs on the initially constructed solution (obtained by the LJFR-SJFR heuristic) is very important. Additionally, it is noticeable in this table the improvement obtained by cMA+LMCTS over the compared algorithms, since it outperforms the compared algorithms for all considered instances. The exception are the inconsistent instances, for which the TS algorithm is the best one. The other proposed cMA, cMA+LTH, which obtained the best results for the makespan value is worse than both cMA+LMCTS and the Struggle GA for the flowtime objective.

10.5 Computational Results on Dynamic Instances

The study made in Section 10.4 using static instances for the problem of resource allocation in grids allowed us to better know the behavior of the cMAs, showing their main differences in the resolution of the problem and the results we could expect from them for several different cases. However, even if we can define static instances with really complex features, we still need to analyze the behavior of

the algorithms in a more realistic dynamic grid system environment. In this case, the algorithm typically has to schedule the tasks in very short time intervals, and in a dynamic scenario that is continuously changing with time (resources that join and leave the Grid system). Thus, we study in this section the behavior of our algorithms in a more realistic set of dynamic instances. These instances are obtained using a simulator of a grid environment proposed in [24] that allows us to simulate different grid environments with distinct parameterizations. This simulator is briefly described in Section 10.5.1.

10.5.1 Dynamic Grid Simulator

The dynamic grid simulator was built from the Hypersim [20] framework, which is at the same time based in the simulation of systems of discrete events. The dynamic grid simulator allows us to emulate a set of dynamic resources that appear and disappear along time simulating resources that are registered and unregistered in grids. These simulated resources have different computing capacities (by means of number of instructions per time unit), and there is no limit on the number of tasks that can be assigned to a given resource. Moreover, every resource could have its own local scheduling policy.

New tasks arrive to the system following different distributions. The modelled tasks have intensive computing requirements, and they differ each other only in the work load (number of instructions). Tasks are considered to be sequential and have no dependencies on the other ones, so they are not restricted by the order in which they are executed, and no communication is needed among them. Hence, tasks are run in one single resource, and cannot be interrupted unless there is some error during the run. The scheduling process of these tasks is centralized, allowing to compare the scheduling algorithms easier than in the case of a decentralized system, since in this case the result of the scheduling is highly dependent on the the structure defined by the schedulers. The design of this simulator allows to easily adapt different scheduling policies, and it offers already implemented some scheduling policies. Anyway, the simulator is compatible both with static and dynamic schedulers.

The scheduler in our simulated grid is dynamically adapted to the evolution of the grid through the re-scheduling of the tasks either with a given frequency or when a change in the grid resources is made. As it could be expected from a scheduler of a real grid. In our simulator (at least in the version we are using in this work), no possible dependencies are considered between tasks and resources, so tasks can be run in any resource, and the computation time depends on both the length of the task and the resource capacity.

Finally, this simulator provides a configurable environment that allows the user to define different grid scenarios simply by changing some parameters. The simulator provides a large number of statistical measures that allows the user to evaluate and compare different schedulers, as well as the influence of the different parameter values.

10.5.2 Dynamic Benchmark Description

In this section, we present the parametrization used for the simulator described in Section 10.5.1 in order to define the benchmark for testing our schedulers. This parametrization have been carefully set in order to have different kinds of real grids. This way, we have defined grids of different (random) sizes, that we have enclosed in four different sets called small, medium, large, and very large, having the resources composing these grids random computing capacities. The details on the parametrization of the used simulator are given in Table 10.5, and the meaning of every parameter in the table is explained next:

- *Init. hosts*: Number of resources initially in the environment.
- *Max. hosts*: Maximum number of resources in the grid system.
- *Min. hosts*: Minimum number of resources in the grid system.
- *MIPS*: Normal distribution modelling computing capacity of resources.
- *Add host*: Normal distribution modelling the frequency of new resources being added to the system.
- *Delete host*: Normal distribution modelling the frequency of resources being dropped from the system.
- *Total tasks*: Number of tasks to be scheduled.
- *Init. tasks*: Initial number of tasks in the system to be scheduled.
- *Workload*: Normal distribution modelling the workload of tasks.
- *Interarrival*: Frequency (given by an exponential distribution) of new tasks arriving to the system (it is ensured that each time the simulator is activated, there will be at least one new task per resource).
- *Activation*: Establishes the activation policy according to an exponential distribution.
- *Reschedule*: When the scheduler is activated, this parameter indicates whether the already assigned tasks, which have not yet started their execution, will be rescheduled.

Table 10.5. Settings for the dynamic grid simulator

	Small	**Medium**	**Large**	**Very Large**
Init. hosts	32	64	128	256
Max. hosts	37	70	135	264
Min. hosts	27	58	121	248
MIPS	$N(1000, 175)^*$			
Add host	$N(625000, 93750)$	$N(562500, 84375)$	$N(500000, 75000)$	$N(437500, 65625)$
Delete host	$N(625000, 93750)$			
Total tasks	512	1024	2048	4096
Init. tasks	384	768	1536	3072
Workload	$N(2.5 * 10^8, 4.375 * 10^7)$			
Interarrival	$E(7812.5)^\dagger$	$E(3906.25)$	$E(1953.125)$	$E(976.5625)$
Activation	Resource_and_time_interval(250000)			
Reschedule	True			
Host select	All			
Task select	All			
Number of runs	15			

$^*N(\mu, \sigma)$ is a uniform distribution with average value μ and standard deviation σ.
$^\dagger E(\mu)$ is an exponential distribution an average value μ.

- *Host selection*: Selection policy of resources (*all* means that all resources of the system are selected for scheduling purposes).
- *Task selection*: Selection policy of tasks (*all* means that all tasks in the system must be scheduled).
- *Number runs*: Number of simulations done with the same parameters. Reported results are then averaged over this number.

As it can be seen in Table 10.5, we have defined four different grid sizes for our studies. Small grids are composed of a maximum of 37 hosts and a minimum of 27. The initial value is set to 32 and then it dynamically changes in that interval. When the simulation starts, 384 tasks must be scheduled, and new tasks arrive along time until a total of 512 ones. Medium grids are composed by a number of hosts in the interval $[58, 70]$, starting with 64, and the total number of tasks is 1024 (being 768 at the beginning of the simulation). The large grids are considered to have between 121 and 135 hosts (128 initially) and a number of 2048 tasks (starting from 1536). Finally, the largest grids studied in this section are composed by an average of 256 hosts (varying this value in the interval $[248, 264]$), and the number of tasks to schedule grows from 3072 (initial fixed value) up to 4096. In all the configurations, the computing capacity of resources, their frequency of appearing and disappearing, the length of tasks and their arrival frequency are randomly set parameters (with values in the specified intervals). In the rescheduling process, all non executed tasks are considered even if they were previously scheduled.

10.6 Evaluation and Discussion

We proceed in this section to evaluate our schedulers in the dynamic benchmark previously defined. In order to make more realistic simulations, we have reduced the run time for the algorithm from 90 to 25 seconds, and we run the algorithm in a a Pentium IV 3.5GHz with 1GB RAM under windows XP operating system without any other process in background. The parametrization of the algorithm is shown in Table 10.6. As it can be seen, there are some small differences between this configuration and the one used in Section 10.4. These changes were made in order to promote the exploration capabilities of the algorithm for this set of more complex instances, and improve its answer to the instance changes. Thus, we have increased the population size to 6×6 (instead of 5×5) for solving all the instances except the small ones, the neighborhood is changed to L5 instead of C9, and we also improved the generation of the initial population. In the case of Section 10.4, this was made by generating one first individual using the LJFR-SJFR method, and the other individuals of the population were obtained after applying strong mutations to this initial individual. In this case, two initial individuals are generated instead of one: one with LJFR-SJFR, and the other one using the minimum completion time method (MCT). Then, the rest of the population is generated by mutating one of these two initial individuals (selected with equal probability).

Table 10.6. Parameterization of cMA+LMCTS for the dynamic benchmark

Termination condition	25 seconds run or $2 \times nb\_tasks$ generations
Population size	5×5 (small grids)
	6×6 (medium, large, and very large grids)
Probability of recombination	$p_c = 1.0$
Probability of mutation	$p_m = 0.5$
Population initialization	MCT and LJFR-SJFR
Neighborhood pattern	L5
Recombination order	FLS (Fixed Line Sweep)
Mutation order	NRS (New Random Sweep)
Selection method	3-Tournament
Recombination operator	One-Point recombination
Mutation operator	Rebalance
Local search method	LMCTS
Number of iterations of the local search	5
Replacement policy	Replace if better

The MCT method assigns a job to the machine yielding the earliest completion time (the ready times of the machines are used). When a job arrives in the system, all available resources are examined to determine the resource that yields the smallest completion time for the job (note that a job could be assigned to a machine that does not have the smallest execution time for that job). This method is also known as Fast Greedy, originally proposed for SmartNet system [14].

The parametrization for cMA+LTH is the same one proposed for cMA+LMCTS in Table 10.6 with only one exception: the population is set to 3×3 as an attempt to reduce the computational overload introduced by the expensive LTH method.

In our tests, the algorithms were run for 30 independent runs. The results are given in tables 10.7 and 10.8 for the makespan and the flowtime, respectively. Specifically, we present the average in the 30 runs of the average value (for the makespan and the flowtime, respectively) during every run (this value is continuously changing during the run due to the grid dynamism), the standard deviation (with a 95% confidence interval –CI–), and the deviation between the current solution and the best one for the same instance size. A 95% CI means that we can be 95% sure that the range of makespan (flowtime) values are within the shown interval, if the experiment were to run again. We are comparing in these tables the results obtained by our two cMAs and the same algorithms but with a panmictic (non structured) population, which are the best results we found in the literature for the studied problems [25].

In Table 10.7 we can see that the best overall algorithm for the four kinds of instances is cMA+LTH, which is the best algorithm in three out of the four cases (best values for every instance size are **bolded**). Only in the case of the largest instances cMA+LTH is outperformed by another algorithm (namely MA+LTH), but it is the second best algorithm in this case. Regarding the local search method used, we obtain from the results that the algorithms using TS as a local search method (MA+LTH and cMA+LTH) clearly outperform the other two ones for the four different instance sizes, since these two algorithms are the best ones for the four instances.

Table 10.7. Makespan values for the dynamic instances

	Heuristic	Makespan	% CI (0.95)	Best Dev.
Small	MA+LMCTS	4161118.81	1.47%	0.34%
	MA+LTH	4157307.74	1.31%	0.25%
	cMA+LMCTS	4175334.61	1.45%	0.68%
	cMA+LTH	**4147071.06**	**1.33%**	**0.00%**
Medium	MA+LMCTS	4096566.76	0.94%	0.32%
	MA+LTH	4083956.30	0.70%	0.01%
	cMA+LMCTS	4093488.97	0.71%	0.25%
	cMA+LTH	**4083400.11**	**0.62%**	**0.00%**
Large	MA+LMCTS	4074842.81	0.69%	0.29%
	MA+LTH	4067825.95	0.77%	0.12%
	cMA+LMCTS	4087570.52	0.57%	0.60%
	cMA+LTH	**4063033.82**	**0.49%**	**0.00%**
Very Large	MA+LMCTS	4140542.54	0.80%	0.82%
	MA+LTH	**4106945.59**	**0.74%**	**0.00%**
	cMA+LMCTS	4139573.56	0.35%	0.79%
	cMA+LTH	4116276.64	0.72%	0.23%

Table 10.8. Flowtime values for the dynamic instances

	Heuristic	Flowtime	% CI (0.95)	Best Dev.
Small	MA+LMCTS	1045280118.16	0.93%	0.15%
	MA+LTH	1045797293.10	0.93%	0.20%
	cMA+LMCTS	**1044166223.64**	**0.92%**	**0.00%**
	cMA+LTH	1046029751.67	0.93%	0.22%
Medium	MA+LMCTS	2077936674.17	0.61%	0.07%
	MA+LTH	2080903152.40	0.62%	0.22%
	cMA+LMCTS	**2076432235.04**	**0.60%**	**0.00%**
	cMA+LTH	2080434282.38	0.61%	0.19%
Large	MA+LMCTS	4146872566.09	0.54%	0.02%
	MA+LTH	4153455636.89	0.53%	0.18%
	cMA+LMCTS	**4146149079.39**	**0.55%**	**0.00%**
	cMA+LTH	4150847781.82	0.53%	0.11%
Very Large	MA+LMCTS	**8328971557.96**	**0.35%**	**0.00%**
	MA+LTH	8341662800.11	0.35%	0.15%
	cMA+LMCTS	8338100602.75	0.34%	0.11%
	cMA+LTH	8337173763.88	0.35%	0.10%

The results obtained for the flowtime are given in Table 10.8. As it happened in the previous case, the best cMA (cMA+LMCTS in this case) outperforms the best MA (MA+LMCTS) algorithm for all the tested instance sizes, with the only exception of the very large one. We notice that in this case, the results are also somehow opposite to the ones obtained for the makespan, since the algorithms implementing the LMCTS local search method outperform those using LTH for all the instances. However, these results make sense, since both makespan and flowtime are conflictive objective values. This means that, for high quality solutions, it is not possible to improve one of the two objectives without decreasing the quality of the other. This is related to the concept of Pareto optimal front in multi-objective optimization (see [12,13]). In this paper we are tackling a multi-objective problem by weighting the two objectives into a single fitness function. Thus, in this work we are giving more importance to the makespan objective by weighting this value by 0.75 in the fitness function, while the weight of flowtime

was set to 0.25. So we can consider that cMA+LTH is the algorithm obtaining the best overall results among the tested ones.

10.7 Conclusions and Future Work

In this work we have presented two implementations of Cellular Memetic Algorithms (cMAs) for the problem of job scheduling in Computational Grids when both makespan and flowtime are simultaneously minimized. cMAs are a family of population-based metaheuristics that have turned out to be an interesting approach due to their structured population, which allows to better control the tradeoff between the exploitation and exploration of the search space. We have implemented and experimentally studied several methods and operators of cMA for the job scheduling in Grid systems, which is a challenging problem in today's large-scale distributed applications.

The proposed cMAs were tested and compared versus other algorithms in the literature for benchmarks using both static and dynamic instances. Our experimental study showed that cMAs are a good choice for scheduling jobs in Computational Grids given that they are able to deliver high quality planning in a very short time. This last feature makes cMAs useful to design efficient dynamic schedulers for real Grid systems, which can be obtained by running the cMA-based scheduler in batch mode for a very short time to schedule jobs arriving in the systems since the last activation of the cMA scheduler. The use of the proposed cMA could highly improve the behavior of real clusters in which very simple methods (e.g., queuing systems or *ad hoc* schedulers using specific knowledge of the grid infrastructure) are used.

In our future work we would like to better understand some issues raised by the experimental study such as the good performance of the cMAs for consistent and semi-consistent Grid Computing environments and the not so good performance for inconsistent computing instances. Also, we plan to extend the experimental study by considering other operators and methods as well as studying the performance of cMA-based scheduler(s) in longer periods of time and considering larger grids. Additionally, we are studying different policies for applying the local search method in order to make this important step of the algorithm less computationally expensive. Other interesting line for future research is to tackle the problem with a multi-objective algorithm in order to find a set of non-dominated solutions to the problem.

Acknowledgments

F. Xhafa acknowledges partial support by Projects ASCE TIN2005-09198-C02-02, FP6-2004-ISO-FETPI (AEOLUS) and MEC TIN2005-25859-E and FORMALISM TIN2007-66523. E. Alba acknowledges that this work has been partially funded by the Spanish MEC and FEDER under contract TIN2005-08818-C04-01 (the OPLINK project).

References

1. Abraham, A., Buyya, R., Nath, B.: Nature's heuristics for scheduling jobs on computational grids. In: The 8th IEEE International Conference on Advanced Computing and Communications (ADCOM), India, pp. 45–52. IEEE Press, Los Alamitos (2000)
2. Alba, E., Dorronsoro, B.: The exploration/exploitation tradeoff in dynamic cellular evolutionary algorithms. IEEE Transactions on Evolutionary Computation 9(2), 126–142 (2005)
3. Alba, E., Dorronsoro, B.: Cellular Genetic Algorithms. In: Operations Research/-Computer Science Interfaces. Springer, Heidelberg (to appear)
4. Alba, E., Dorronsoro, B., Alfonso, H.: Cellular memetic algorithms. Journal of Computer Science and Technology 5(4), 257–263 (2005)
5. Alba, E., Dorronsoro, B., Alfonso, H.: Cellular memetic algorithms evaluated on SAT. In: XI Congreso Argentino de Ciencias de la Computación (CACIC) (2005) DVD Edition
6. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation 6(5), 443–462 (2002)
7. Alba, E., Troya, J.M.: Cellular evolutionary algorithms: Evaluating the influence of ratio. In: Deb, K., Rudolph, G., Lutton, E., Merelo, J.J., Schoenauer, M., Schwefel, H.-P., Yao, X. (eds.) PPSN 2000. LNCS, vol. 1917, pp. 29–38. Springer, Heidelberg (2000)
8. Alba, E., Dorronsoro, B., Giacobini, M., Tomassini, M.: Handbook of Bioinspired Algorithms and Applications. In: Decentralized Cellular Evolutionary Algorithms, ch. 7, pp. 103–120. CRC Press, Boca Raton (2006)
9. Braun, H., Siegel, T.D., Beck, N., Bölöni, L., Maheswaran, M., Reuther, A., Robertson, J., Theys, M., Yao, B.: 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)
10. Carretero, J., Xhafa, F.: Using genetic algorithms for scheduling jobs in large scale grid applications. Journal of Technological and Economic Development –A Research Journal of Vilnius Gediminas Technical University 12(1), 11–17 (2006)
11. Casanova, H., Legrand, A., Zagorodnov, D., Berman, F.: Heuristics for scheduling parameter sweep applications in grid environments. In: Heterogeneous Computing Workshop, pp. 349–363 (2000)
12. Coello, C.A., Van Veldhuizen, D.A., Lamont, G.B.: Evolutionary Algorithms for Solving Multi-Objective Problems. In: Genetic Algorithms and Evolutionary Computation. Kluwer Academic Pubishers, Dordrecht (2002)
13. Deb, K.: Multi-Objective Optimization using Evolutionary Algorithms. Wiley, Chichester (2001)
14. Freund, R., Gherrity, M., Ambrosius, S., Campbell, M., Halderman, M., Hensgen, D., Keith, E., Kidd, T., Kussow, M., Limaand, J., Mirabile, F., Moore, L., Rust, B., Siegel, H.J.: Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In: Seventh Heterogeneous Computing Workshop, pp. 184–199 (1998)
15. Ghafoor, A., Yang, J.: Distributed heterogeneous supercomputing management system. IEEE Comput. 26(6), 78–86 (1993)
16. Giacobini, M., Tomassini, M., Tettamanzi, A.G.B., Alba, E.: Selection intensity in cellular evolutionary algorithms for regular lattices. IEEE Transactions on Evolutionary Computation 9(5), 489–505 (2005)

17. Glover, F., Laguna, M.: Tabu Search. Kluwer Academic Publishers, Boston (1997)
18. Kafil, M., Ahmad, I.: Optimal task assignment in heterogeneous distributed computing systems. IEEE Concurrency 6(3), 42–51 (1998)
19. Luna, F., Nebro, A.J., Alba, E.: Observations in using grid-enabled technologies for solving multi-objective optimization problems. Parallel Computing 32, 377–393 (2006)
20. Phatanapherom, S., Kachitvichyaunukul, V.: Fast simulation model for grid scheduling using hypersim. In: Proceedings of the 2003 Winter Simulation Conference, pp. 1494–1500 (2003)
21. Talbi, E.-G.: Parallel Combinatorial Optimization. John Wiley & Sons, USA (2006)
22. Talbi, E.-G., Zomaya, A.: Grids for Bioinformatics and Computational Biology. John Wiley & Sons, USA (2007)
23. Xhafa, F., Carretero, J., Alba, E., Dorronsoro, B.: Design and Evaluation of Tabu Search Method for Job Scheduling in Distributed Environments. In: The 11th International Workshop on Nature Inspired Distributed Computing (NIDISC 2008) held in conjunction with The 22th IEEE/ACM International Parallel and Distributed Processing (NIDISC 2008), Florida, USA, April 14-18 (to appear, 2008)
24. Xhafa, F., Carretero, J., Barolli, L., Durresi, A.: Requirements for an event-based simulation package for grid systems. Journal of Interconnection Networks 8(2), 163–178 (2007)
25. Xhafa, F.: Hybrid Evolutionary Algorithms. In: A Hybrid Heuristic for Job Scheduling in Computational Grids, ch. 11. Studies in Computational Intelligence, vol. 75, pp. 269–311. Springer, Heidelberg (2007)
26. Xhafa, F.: 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 (2006)