View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

GENETIC ALGORITHM BASED SCHEDULERS FOR GRID COMPUTING SYSTEMS

Fatos Xhafa Javier Carretero

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Campus Nord - Ed. Omega, C/Jordi Girona Salgado 1-3
08034 Barcelona, Spain
fatos@lsi.upc.edu

Ajith Abraham

IITA Professorship Program, School of Computer Science
Yonsei University
Sudaemoon-ku, Seoul 120-749, Republic of Korea
ajith.abraham@ieee.org

Abstract. *In this paper we present Genetic Algorithms (GAs) based schedulers for efficiently allocating jobs to resources in a Grid system. Scheduling is a key problem in emergent computational systems, such as Grid and P2P, in order to benefit from the large computing capacity of such systems. We present an extensive study on the usefulness of GAs for designing efficient Grid schedulers when makespan and flowtime are minimized. Two encoding schemes has been considered and most of GA operators for each of them are implemented and empirically studied. The extensive experimental study showed that our GA-based schedulers outperform existing GA implementations in the literature for the problem and also revealed their efficiency when makespan and flowtime are minimized either in a hierarchical or a simultaneous optimization mode; previous approaches considered only the minimization of the makespan. Moreover, we were able to identify which GAs versions work best under certain Grid characteristics, which is very useful for real Grids. Our GA-based schedulers are very fast and hence they can be used to dynamically schedule jobs arrived in the Grid system by running in batch mode for a short time.*
Keywords: Computational Grids, Scheduling, Genetic Algorithms, Resource Allocation, Makespan, Flowtime, Expected Time to Compute, Benchmark Simulation Model.

1. **Introduction.** A computational grid is a large scale, heterogeneous collection of autonomous systems, geographically distributed and interconnected by heterogeneous networks. Job sharing (computational burden) is one of the major difficult tasks in a computational grid environment. Grid resource manager provides the functionality for discovery and publishing of resources as well as scheduling, submission and monitoring of jobs. However, computing resources are geographically distributed under different ownerships each having their own access policy, cost and various constraints. Since the introduction of computational grids by Foster et al. [11, 12], this problem is increasingly receiving the attention of researchers due to the use of Grid infrastructures in solving complex problems from many fields of interest such as optimization, scientific simulation, drug discovery, bio-informatics etc. Unlike scheduling problems in conventional distributed systems, this

problem is much more complex as new features of Grid systems such as its dynamic nature and the high degree of heterogeneity of jobs and resources must be tackled. The problem is multi-objective in its general formulation, the two most important objectives being the minimization of makespan and flowtime of the system.

Job scheduling is known to be NP-complete [13], therefore the use of heuristics is the *de facto* approach in order to cope in practice with its difficulty. Thus, the meta-heuristics computing research community has already started to examine this problem. Single heuristic approaches for the problem include Local Search (Ritchie and Levine [22]), Simulated Annealing (Yarkhan and Dongarra [25], Abraham et al. [1]) and Tabu Search (Abraham et al. [1]). GAs for scheduling are addressed in several works (Braun et al. [5], Zomaya and Teh [26], Martino and Mililotti [19], Abraham et al. [1], Page and Naughton [20]). Some hybrid heuristic approaches have also been reported for the problem. Thus, Abraham et al. [1] addressed the hybridization of GA, SA and TS heuristics for dynamic job scheduling on large-scale distributed systems. In these hybridizations a population-based heuristic, such as GAs, is combined with two other Local Search heuristics, such as TS and SA, that deal with only one solution at a time. Ritchie and Levine [21, 23] combined an Ant Colony Optimization algorithm with a TS algorithm for the problem. Other approaches for the problem include the use of AI techniques (Cao et al. [8]), use of predictive models to schedule jobs at both system and application level (Gao et al. [14]), Particle Swarm Optimization (Abraham et al. [2]), Fuzzy based scheduling (Kumar et al. [17]) and economic-based approaches (Buyya et al. [7], Abraham et al. [3] and Buyya [6]).

In this work several GAs are proposed and efficiently implemented in C++ using a generic approach based on a skeleton for GAs [4]. Two encoding schemes has been considered and most of GA operators for each of them are implemented and empirically studied. The implementation has been extensively tested, on the one hand, to identify a set of appropriate values for the parameters that conduct the search and, on the other, to compare the results with the best known results for the problem in the evolutionary computing literature. We have used the benchmark of Braun et al. [5], which consists of instances that try to capture different degrees of heterogeneity of Grid resources[1] and workload of jobs and resources. The experimental study aimed at revealing the efficiency of GA-based schedulers when makespan and flowtime are both optimized either in a hierarchical optimization (being makespan the primary objective) or in a simultaneous optimization mode; previous GA approaches in the literature considered only the minimization of the makespan. Moreover, in the proposed experimental study we addressed the issue of identifying which GA operators work best under certain Grid characteristics, such as consistency of computing, degree of heterogeneity of resources and jobs. This last feature is useful for designing dynamic schedulers that would adaptively allocate jobs to resources in real Grids.

It should be noted, however, that meta-heuristics run on static instances of the problem and therefore in this approach static schedulers are obtained. In order to deal with the dynamics of the Grid systems, dynamic schedulers run the static GA scheduler in batch mode to schedule jobs arrived in the system since its last activation. Certainly, the efficiency of the heuristic is crucial for the quality of the dynamic scheduler.

[1] We use indistinctly the terms resource/machine.

The remainder of the paper is organized as follows. The problem of job scheduling is described and formally introduced in Section 2. The proposed GAs and their particularization for the problem are given in Section 3. We give an extensive experimental study in Section 4 and summarize in Section 5 most important aspects of this work.

2. **Scheduling in Grid systems.** Due to the complexity of Grid systems and large scale distributed applications, different versions and modes of scheduling can be considered. We will consider here a version of the problem, which doesn't take into account possible restrictions on job interdependencies, data transmission and economic and cost policies on resources. We are essentially concerned here with scheduling version needed to achieve high performance applications and, from the Grid users perspective, to offer QoS of the Grid system. This type of scheduling arises in applications that can be solved by splitting them into many independent jobs, submitting them to the Grid and combining the partial results to obtain the final solution. One such family of applications is that of *parameter sweep* applications [10, 9]. Monte-Carlo simulations, for instance, belongs to this family. Moreover, in Grid systems there is a need for allocating independent user applications to Grid resources. We consider thus the scenario in which jobs submitted to the Grid are independent and are not preemptive (they cannot change the resource they has been assigned to once their execution is started, unless the resource is dropped from the Grid.)

2.1. **Problem formulation.** In order to capture most important characteristics of job scheduling in Grid systems, we considered the use of a simulation model rather than existing models of real Grid systems. To formulate the problem under such simulation model, an estimation of the computational load of each job, the computing capacity of each resource, and an estimation of the prior load of each one of the resources are required. One such model is the *Expected Time to Compute (ETC)* model [18, 5]. Thus we make the usual assumption that we know the computing capacity of each resource, an estimation or prediction of the computational needs (*workload*) of each job, and the load of prior work of each resource. Having the computing capacity of the resources and the workload of the jobs, an Expected Time to Compute matrix ETC can be built, where each position $ETC[t][m]$ indicates the expected time to compute job t in resource m. The entries $ETC[t][m]$ could be computed by dividing the workload of job t by the computing capacity of resource m, or in more complex ways by including the associated migration cost of job t to resource m, etc. It is realistic to assume that this formulation is applicable in practice, since it is easy to know the computing capacity of each resource and the requirements about computation need of the jobs can be known from specifications provided by the user, from historic data or from predictions. Examples of computation of the workload of jobs are known from the Cornell Theory Center [16].

Interestingly, the ETC matrix model allows to capture important characteristics of job scheduling. For instance, the ETC model allows to quite easily introduce possible inconsistencies among jobs and resources in the Grid system by giving a large value to $ETC[t][m]$ to indicate that job t is incompatible with resource m. Moreover, benchmarks of instances are generated from this model [5] capturing different characteristics of distributed heterogeneous systems such as consistency of computing, heterogeneity of resources and heterogeneity of jobs (see Section 4 for more details).

We can now formally define an instance of the problem, under the ETC matrix model, as follows. It consists of:

- A *number* of independent *jobs* that must be scheduled. Any job has to be processed entirely in unique resource.
- A *number* of heterogeneous *machines* candidates to participate in the planning.
- The *workload* (in millions of instructions) of each job.
- The *computing capacity* of each machine (in *mips*).
- The time $ready_m$ when the machine m will have finished the previously assigned jobs. This parameter measures the previous workload of a machine.
- The expected time to compute ETC matrix of size $nb_jobs \times nb_machines$, where a position $ETC[t][m]$ indicates the expected execution time of job t in machine m.

2.2. **Fitness.** Several optimization criteria can be considered for this problem, certainly the problem is multiobjective in its general formulation. The fundamental criterion is that of minimizing the *makespan*, that is, the time when finishes the latest job. A secondary criterion is to minimize the *flowtime*, that is, minimizing the sum of finalization times of all the jobs. These two criteria are defined as follows:

$$\text{makespan}: \min_{S_i \in Sched} \{ \max_{j \in Jobs} F_j \} \quad \text{and} \quad \text{flowtime}: \min_{S_i \in Sched} \{ \sum_{j \in Jobs} F_j \}, \tag{1}$$

where F_j denotes the time when job j finalizes, $Sched$ is the set of all possible schedules and $Jobs$ the set of all jobs to be scheduled. Note that makespan is not affected by any particular execution order of the jobs in a concrete resource, while in order to minimize flowtime of a resource, jobs should be executed in a ascending order of their expected time to compute. Essentially, we want to maximize the productivity (*throughput*) of the Grid through an intelligent load balancing and at the same time we want to obtain planning that offer an acceptable *QoS*. It should also be noted that makespan and flowtime are contradictory objectives; trying to minimize one of them could not suit to the other, especially for planning close to optimal ones.

In order to express the Eq. (1) in an easily computable form, we express makespan in terms of the *completion time* of a machine. The intuition behind this is that the time when finishes the last job equals the completion time of the last machine in completing its jobs. Let *completion* be a vector of size $nb_machines$, in which $completion[m]$ indicates the time in which machine m will finalize the processing of the previous assigned jobs and of those already planned for the machine. Thus, $completion[m]$ is calculated as follows:

$$completion[m] = ready_times[m] + \sum_{\{j \in Jobs \ | \ schedule[j]=m\}} ETC[j][m]. \tag{2}$$

Then, makespan from Eq. (1) can be now expressed as:

$$makespan = \max\{completion[i] \mid i \in Machines\}. \tag{3}$$

These criteria can be integrated in several ways to establish the desired priority among them. In the multi-objective optimization two fundamental models are the hierarchical and the simultaneous approach. In the former, the optimization criteria are sorted by their importance, in a way that if a criterion c_i is of smaller importance than criterion c_j,

the value for the criterion c_j cannot be varied while optimizing according to c_i. In the latter approach, an optimal planning is that in which any improvement with respect to a criterion causes a deterioration with respect to the other criterion.

Both approaches are considered in this work. In the hierarchical approach the criterion with more priority is *makespan* and the second criterion is $flowtime$. In the simultaneous approach, makespan and flowtime are minimized simultaneously. In this later case, we have used a weighted sum function since makespan and flowtime are measured in the same unit (arbitrary time units). However, the makespan and flowtime values are in incomparable ranges, due to the fact that flowtime has a higher magnitude order over makespan, and its difference increases as more jobs and machines are considered. For this reason, the value of mean flowtime, $flowtime/nb_machines$, is used to evaluate flowtime. Additionally, both values are weighted in order to balance their importance. Fitness value is thus calculated as:

$$fitness = \lambda \cdot makespan + (1 - \lambda) \cdot mean_flowtime, \quad (4)$$

where λ has been *a priori* fixed after a preliminary tuning process (see Section 4).

3. GAs for Scheduling in Grid Systems.

GAs are high level algorithms that integrate other methods and genetic operators, therefore, in order to implement it for a concrete problem, one just needs to start from a *template* for the method and design the inner methods, operators and appropriate data structures. For the purpose of this work we have used the *template* given in Algorithm 1.

begin
  Initialization: `Generate the initial population` $P(t=0)$ `of` n `individuals`
  Fitness: `Evaluate the fitness of each individual of the population.`
    `Evaluate(`$P(t)$`)`
  while `(not termination condition)` **do**
    Selection: `Select a subset of` m `pairs from` $P(t)$`. Let` $P_1(t)$ `= Select(`$P(t)$`).`
    Crossover: `With probability` p_c`, cross each of the` m `chosen pairs.`
      `Let` $P_2(t)$ `= Cross(`$P_1(t)$`) be the set of offsprings.`
    Mutation: `With probability` p_m`, mutate each offspring in` $P_2(t)$`.`
      `Let` $P_3(t)$ `= Mutate(`$P_2(t)$`).`
    Fitness: `Evaluate the fitness of each offspring. Evaluate(`$P_3(t)$`).`
    Replacement: `Create a new generation from individuals in` $P(t)$ `and` $P_3(t)$`.`
      `Let` $P(t+1)$ `= Replace(`$P(t), P_3(t)$`);` $t = t + 1$`.`
  fwhile
  return `Best found solution;`
end

FIGURE 1. Genetic Algorithm template

3.1. Schedule encodings.

The encoding of individuals (also known as *chromosome*) of the population is a key issue in evolutionary-like algorithms. Note that for a combinatorial optimization problem, individual refers to a solution of the problem. Encodings determine the type of the operators that could be used to ensure the evolution of the individuals

and also impact on the feasibility of individuals. Reproduction operators are applied on the chromosomes, hence appropriate representation and reproduction operators are determinant on the behavior of the evolutionary-like algorithm.

In the literature, different types of representations are reported. We use two of them, namely the *direct representation* and the *permutation-based representation*.

Direct representation. For job scheduling problem, a direct representation is obtained as follows. Feasible solutions are encoded in a vector, called *schedule*, of size *nb_jobs*, where $schedule[i]$ indicates the machine where job i is assigned by the schedule. Thus, the values of this vector are natural numbers included in the range $[1, nb_machines]$. Note that in this representation a machine number can appear more than once.

Permutation-based representation. Unlike the direct representation where elements representing number of machines can be repeated, in this representation each element must be present only once. This kind of representation is especially useful in sequence-based problems, thus it is also interesting for scheduling problems. For the job scheduling problem this representation is obtained in two steps: (a) for each machine m_i, construct the sequence S_i of jobs assigned to it; and (b) concatenate sequences S_i. The resulting sequence is a permutation of jobs assigned to machines.

Note that this representation requires maintaining additional information on the number of jobs assigned to each machine. This can be easily done by maintaining a vector of size equal to *nb_machines* whose components indicate, for any machine, the number of jobs assigned to that machine.

One advantage of the direct representation is that it defines a feasible schedule in a straightforward way while for the permutation-based representation, the schedule should be "extracted" from the representation. In order to avoid keeping two different representation, we notice that there is a simple transformation from direct representation to permutation-based one and vice-versa. This transformation is achieved in $O(nb_jobs)$ time. Therefore, by using the transformation, we just implemented the direct representation and move efficiently back and forth from one representation to the other.

3.2. **GA's methods and operators.** Next we detail the particularization of methods and operators of the GA template (see Fig. 1) for the scheduling problem based on the two schedule representations presented above.

3.2.1. *Generating the initial population.* In GAs, the initial population is usually generated randomly. Besides the random method, we have used two *ad hoc* heuristics, namely the *Longest Job to Fastest Resource - Shortest Job to Fastest Resource* (LJFR-SJFR) heuristic [1] and *Minimum Completion Time* (MCT) heuristics [18]. These two methods are aimed at introducing more diversity to the initial population.

The LJFR-SJFR heuristic tries to optimize alternatively both values of makespan and flowtime. LJFR-SJFR initially assigns the *nb_machines* longest jobs, increasingly sorted by their workload, to the *nb_machines* available resources, increasingly sorted by their computing capacity (application of LJFR heuristic). Then, for the remaining jobs, in every step, an unassigned job j is assigned to the first available machine, being j the longest job (LJFR) or the shortest job (SJFR) alternatively. This is done until all jobs have been allocated.

In the MCT method, each job is assigned to the machine in which the job obtains the minimum completion time (note that this is not the same as the minimum execution time!). Jobs are considered for allocation at random.

3.3. **Fitness.** See Subsection 2.2 (Problem formulation).

3.4. **Crossover operators.** The crossover operators are the most important ingredient of any evolutionary-like algorithm. Indeed, by selecting individuals from the parental generation and interchanging their *genes*, new individuals (descendants) are obtained. The aim is to obtain descendants of better quality that will feed the next generation and enable the search to explore new regions of solution space not explored yet.

There exist many types of crossover operators explored in the evolutionary computing literature, which depends on the chromosome representation. Thus, in our case, we have crossover operators for the direct representation and other crossover operators for the permutation-based representation.

3.4.1. *Crossover operators for direct representation.* Several crossover operators have been considered for the direct representation.

One-point crossover: Given two parent solutions, this operator, first chooses randomly a position between 1 and nb_jobs. The resulting position serves as a "cutting point" splitting each parent into two segments. Then, the two first parts of the parents are interchanged yielding two new descendants. Note that this operator shows a positional bias, namely, genes that are close together on the chromosome are more probable to be passed to the descendant while genes located at either end of the chromosome will not be recombined.

Two-point crossover: This operator is a generalization of one-point crossover. Now, two cutting points are randomly chosen splitting thus each parent into three segments; the middle parts are interchanged yielding two new descendants. It should be noted that despite of having two cutting points, this operator still shows a positional bias. Notice that it is also possible to define the generalized $k-$point crossover, for $k \geq 3$; for such values this operator tends to explore more thoroughly the solution space, however, it is very likely that it will "destroy" the structure of the parents.

Uniform crossover: This operator generates a *crossover mask* and uses it to generate one descendant. In our case, having chromosomes of length nb_jobs, the crossover mask can be defined as: $crossover\ mask = b_1, b_2, \ldots, b_{nb_jobs}$, where $b_i = 0/1$. Once the mask is generated, for instance by just flipping a fair coin for any position in the chromosome, the descendant $schedule$ is computed from two parents $schedule_1$ and $schedule_2$ as follows:

$$\forall\ i = 1 \ldots nb_jobs, schedule[i] = \begin{cases} schedule_1[i], & \text{if } b_i = 1; \\ schedule_2[i], & \text{if } b_i = 0. \end{cases}$$

It should be noticed that this operator avoids the position bias of previous operators.

Fitness-based crossover: The crossover operators introduced so far do not take into account the fitness of the parents, this is done in the fitness-based crossover in which the crossover mask is computed as follows. For all $i = 1 \ldots nb_jobs$, $schedule[i] = schedule_1[i]$, iff $schedule_1[i] = schedule_2[i]$, otherwise

$$schedule[i] = \begin{cases} schedule_1[i], & \text{with probability } p = f_2/(f_1+f_2); \\ schedule_2[i], & \text{with probability } 1-p, \end{cases}$$

where $f_1 = fitness(schedule_1)$ and $f_2 = fitness(schedule_2)$. Note that if the two parent schedules have similar fitness then the genes of the new descendants are calculated with probability $\sim 1/2$, that is the uniform crossover; however, and, this is the case the fitness-based operator is intended for, when there is a large difference in the fitness of the two parent schedules, then it is quite probable that a chromosome of new structure will be obtained.

3.4.2. *Crossover operators for permutation-based representation.* Crossover operators for permutation representation [26] can be applied to this problem by transforming the direct representation into a permutation-based one. The crossover operators described above for the direct representation are not valid since they often lead to illegal representations. We considered the operators given below.

Partially Matched Crossover (PMX): A segment of genes of one of the parents, say $schedule_1$, is corresponded with the same segment of the other father $schedule_2$ (bi-univoc correspondence). Next, the segment of $schedule_1$ is interchanged with the one of $schedule_2$. Following, the ith descendant ($i = 1, 2$) is filled up by copying the elements of ith father. In case that a job is already present in the descendant, it is replaced according to the correspondence previously described. In case that the correspondence leads to a gene already copied, the other correspondence is considered.

Cycle Crossover (CX): The basic idea of this operator is that each job must occupy the same position, so that only interchanges between *alleles* (positions) will be made. The algorithm works in two steps. First, a cycle of alleles is identified. A cycle of alleles is constructed in the following way: beginning by the first position (allele) of $schedule_1$, we look at the same allele, say a, in $schedule_2$. Then, we go to the allele of $schedule_1$ that contains the gene contained in allele a; the allele is added to the cycle. This step is iteratively repeated until the first allele of $schedule_1$ is reached. The second step, consists of leaving the alleles that form the cycle intact, and the content of the rest is interchanged.

Order Crossover (OX): This operator is intended to preserve the relative order of the jobs in the schedule. The operator receives two schedules $schedule_1$ and $schedule_2$ and produces two new offsprings $schedule'_1$ and $schedule'_2$ as follows:

Generation of the first offspring: (a) Choose a segment of genes from $schedule_1$; (b) copy the segment to the first offspring; (c) copy the jobs that are not in the segment, to the first offspring by starting right from cutting point of the copied segment, according to the order of $schedule_2$ and wrapping around at the end. The *generation of the second offspring* is done similarly. This operator is known as OX−1 operator.

Note that once the permutation-based operator is applied, the offsprings are transformed into the direct representation. The crossover operator is applied with a probability p_c, usually high, which is fixed through a fine tuning process.

3.4.3. *Mutation operators.* We defined several mutation operators based on the direct representation of the schedule that take into account the specific need of load balancing of resources. More precisely, we defined the following operators: *Move*, *Swap*, *Move&Swap* and *Rebalancing*.

Move: This operator moves a job from a resource to another one, so that the new machine is assigned to be different. Note that it is possible to reach any solution by applying successive moves from any given solution.

Swap: Considering movements of jobs between machines is effective, but often it turns out more useful to make interchanges of the allocations of two jobs. Clearly, this operator should be applied to two jobs assigned to different machines.

Move & Swap: The mutation by swap hides a problem: the number of jobs assigned to any resource remains inalterable by mutation. A combined operator avoids this problem in the following way: with a probability $p_{m'}$, a mutation move is applied, otherwise a mutation swap is applied. The value of $p_{m'}$ will depend on the difference of behavior of the mutation operators swap and move.

Rebalancing: The idea is to first improve the solution (by rebalancing the machine loads) and then mutate it. Rebalancing is done in two steps: in the first, a machine m, from most overloaded resources is chosen at random; further, we identify two jobs, t and t' such that job t is assigned to another machine m' whose ETC for the machine m is less than or equal to the ETC of job t' assigned to m; jobs t and t' are interchanged (*swap*). In the second step, in case rebalancing was not possible, we try to rebalance by *move*. After this, a mutation (move or swap each with 0.5 probability) is applied.

The mutation is applied with a probability p_c, usually low, which is fixed through a fine tuning process.

3.4.4. *Selection operators.* Selection operators are used to select the individuals to which the crossover operators will be applied. The following selection operators are investigated: *Select Random*, *Select Best*, *Linear Ranking Selection*, *Binary Tournament Selection*, and $N-$*Tournament Selection*. Note that these operators are independent of the representation, hence we refer the reader to standard descriptions of these operators in the GA literature.

3.4.5. *Replacement operators.* The strategies used for the replacement depend on whether a generational replacement will take place or just a portion of the population will be replaced. In the later case, several strategies can be used depending on how is defined the survival of the individuals. We detail next the main strategies we have explored for our scheduling problem. We denote by P the parental generation and P' the new generation of individuals and by μ and λ their sizes, respectively. Depending on the relation between μ and λ, we have:

- $\mu < \lambda$: If the parameter `replace_only_if_better` is `false`, the worst solutions in P are replaced by the best solutions in P', otherwise a population with best among all solutions in $P \cup P'$ will form the new population. Note that in the later case the replacement shows an elitist behavior.
- $\mu \geq \lambda$: If the parameter `replace_generational` is `false`, the replacement is done as in the case of $\mu < \lambda$ (except when `replace_only_if_better` is `false` in which case the new generation is formed by the best individuals in P'.) Otherwise, the new generation is formed by the best individuals in P'.

Thus, depending on the size of P and P' and the values of `replace_only_if_better` and `replace_generational` we could obtain most standard replacement strategies. We have considered then the replacement operators given next.

Generational replacement: This is the simplest replacement strategy: its is assumed that $\lambda = \mu$ and μ individuals of P are replaced by λ individuals of P'.

Elitist generational replacement: The newly generated offsprings, independently of their fitness, together with two best individuals of the parent generation form the new generation ($\lambda = \mu - 2$).

Replace only if better: The new generation is formed by μ best individuals from the parent generation and the new descendants.

Steady State Strategy: Within this strategy there are different ways to choose the new individuals that will form part of the new population. One such way is to choose a small number of good individuals, recombine them and replace the worst individuals of the parental population by the new descendants. One could also consider a pure elitist replacement. Some research works (e.g. [15]) has shown evidence that Steady State strategy tends to produce a premature stagnation of the population, yet it produces a fast reduction in the fitness (fast convergence). This property will be especially interesting to explore for job scheduling problem since we are interested not only in a good global makespan but also in fast reductions of the makespan.

3.4.6. *Implementation issues.* We have implemented the GA operators presented in the previous section by transforming the main entities and methods into C++ classes and methods, adapting the algorithmic skeletons defined in [4] for the evolutionary algorithms. We give in Fig. 2 the design of a skeleton for evolutionary-like algorithms that we have used for the GAs implementation and briefly describe the meaning of the entities in the diagram. The entities of the skeleton are grouped into *Provided* entities, that are independent of the concrete problem being solved; for these entities a complete implementation is provided. These entities are: `Solver`, `Solver_Seq` and `Setup`. `Solver` is an abstract class that implements the main flow of a GA. This class can be instantiated in a sequential mode or in a parallel mode. In this work the sequential mode has been considered yielding, via inheritance, to the `Solver_Seq` class. The class `Setup` is in charge of managing setup parameters for running the GA implementation. The rest of the entities in the diagram are classified as *Required*, namely, these entities must be completed while instantiating the skeleton for a concrete problem. For these entities, such as `Problem` and `Solution` the skeleton offers just their interfaces whose implementation is done according the problem, in our case job scheduling. Thus, `Population` is a *container* of individuals, i.e., solutions, which offers all the necessary methods to manage the population. On the other hand `Solution` is the class in charge of managing solution representations, solution fitness, etc.

4. **Experimental study.** We used a benchmark of static instances for the problem to obtain computational results in order to study the performance of the GA scheduler. The quality of the schedules produced by our GA implementations are compared with the reported results of GA schedulers in the literature [5]. Static instances were also very useful in finding an appropriate combination of operators and parameters that work well in terms of robustness.

4.1. **Tuning of parameters.** We performed a straightforward tuning process, using static instances generated according to the ETC model, to identify appropriate values for both the proper GA parameters and those related to the job scheduling problem. The

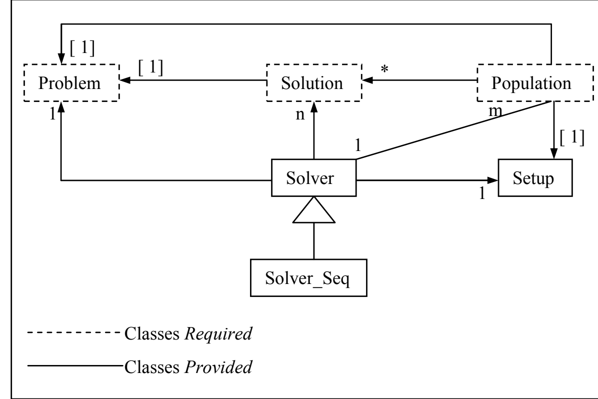

FIGURE 2. The GA skeleton

methodology used for fine tuning tries to independently optimize the parameters according to their priority in the scope of the GA implementation. In the graphs below (Figs. 3 to 7) an average of 20 runs is displayed. In the graphs, y-axis represent the makespan value and x-axis represent the number of GA's generations (number of evolution step). Thus, in each graph we graphically represent the reduction of makespan during the execution of the GA implementation under the corresponding configuration. The value of $\lambda = 0.75$ for the simultaneous approach has been fixed after a preliminary tuning process.

Notice that for the tuning process, we have used randomly generated instances in order to ensure that the parameter setting will be as generic as possible and not related to a concrete benchmark of instances.

Mutation operators. From the tuning of mutation operators, we obtained the values of parameters given in Table 1. For these values, the resulting behavior of mutation operators Move & Swap (three versions) and Rebalancing (five versions) are graphically shown in Fig. 3. The comparison of the performance of the best mutation operators is then presented in Fig. 4. The graphical representation clearly indicates that the worse operator of mutation is *Move* and the best one is *Rebalancing (75%)*. The operator Swap, that a priori seemed to be worse than Move offers a better reduction in makespan, though it is worse than Rebalancing.

TABLE 1. Values of parameters used for comparing the performance of mutation operators

Number of evolution steps	2500
Crossover operator	Two-point CX
Cross probability	0.7
Population size	80
Intermediate population size	78
Selection operator	Select Best
Mutate probability	0.2
Replace generational Replace only if better	False False
Initializing method	LJFR-SJFR

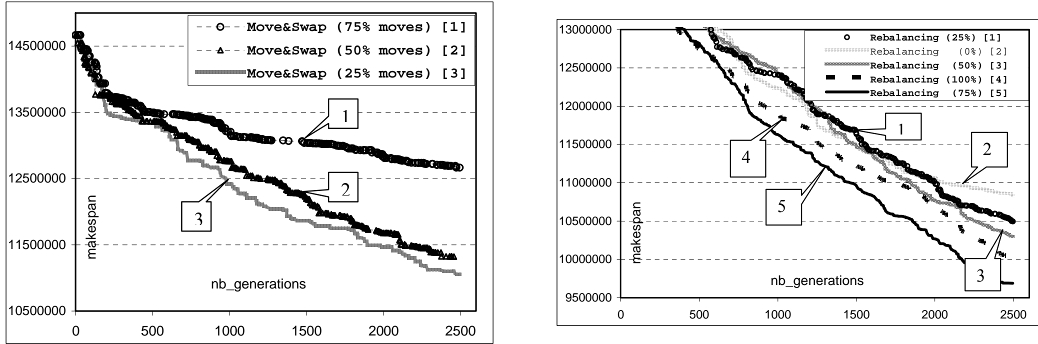

FIGURE 3. Performance of the mutation operator Move&Swap (left) and Rebalancing (right)

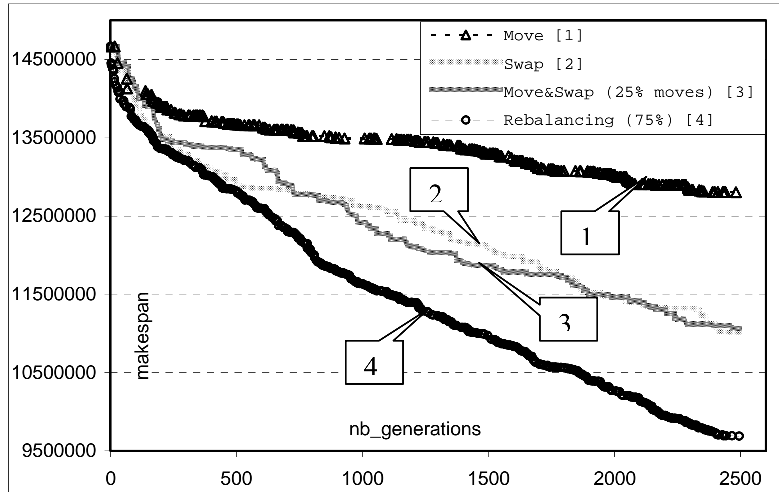

FIGURE 4. Comparison of the performance of mutation operators

Crossover operators. A number of crossover operators were also considered and tuned to find the one of the best behavior; the values for the parameters given in Table 2 were used and their performance is graphically shown in Fig. 5. Observe from Fig. 5 that the operator CX (Cycle Crossover) achieves the best makespan reduction, and with a large difference, among all the considered crossover operators. This is reinforced by the fact that actually the most effective crossover operators in the evolutionary computing literature are permutation-based ones and use precisely this operator [20, 26].

Selection operators. For selection operators we obtained the values for the parameters given in Table 3; their resulting comparison is shown in Fig. 6. It its clear that a suitably adjusted form of tournament selection works best.

Replacement operators. We used the following configuration for tuning replacement operators. Initializing method: LJFR-SJFR, MCT and Random; Selection operator: Linear Ranking; Crossover operator: CX; Mutation operator: Rebalancing 75% and mutate probability=0.3. The rest of parameters are given in Table 4. We notice that replacement operators are known to be computationally expensive. To measure how this would affect

TABLE 2. Parameter values for used for comparing the performance of crossover operators

Number of evolution steps	2500
Cross probability	0.80
Population size	80
Intermediate population size	78
Selection operator	Select Best
Mutation operator	Rebalancing ($p'_m = 0.75$)
Mutate probability	0.2
Replace generational Replace only if better	False False
Initializing method	LJFR-SJFR

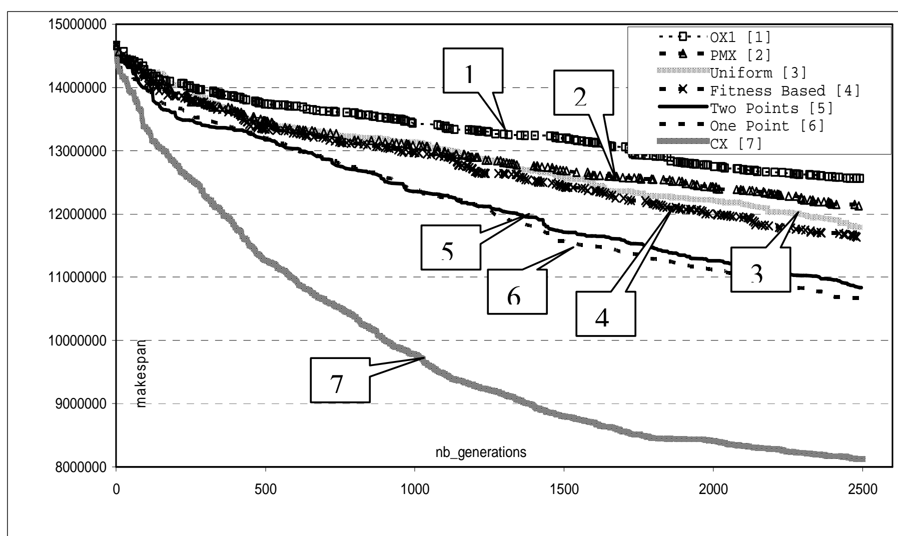

FIGURE 5. Comparison of the performance of crossover operators

TABLE 3. Values of parameters for comparing selection operators

nb_evolution steps	2500
Crossover operator	CX
Cross probability	0.80
Mutation operator	Rebalancing ($p'_m = 0.75$)
Mutate probability	0.2
Population size	80
Intermediate population size	78
Replace generational Replace only if better	False False
Initializing method	LJFR-SJFR

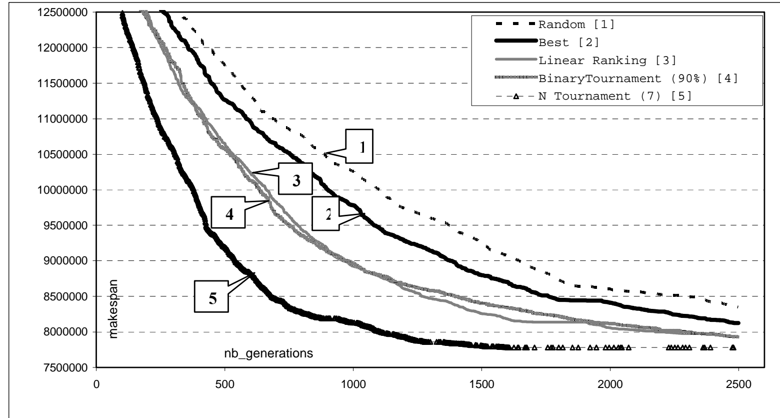

FIGURE 6. Comparison of the performance of different selection operators

the overall execution time of different GAs, we give in the last row of Table 4 values of total amount of work done by the GA algorithm for any replacement operators. Total work W is computed as $W = (p_c + p_m) * pop_size * nb_evolution_steps$. As can be seen from Table 4 the amount of work is pretty much similar for all considered replacement operators.

TABLE 4. Values of parameters for comparing replacement operators

Method / Parameter	Simple Generational	Elitist Generational	Replace If Better	Steady State
Number of evolution steps	2500	2500	2500	16500
Cross probability	0.80	0.80	0.80	1.0
Population size	80	80	80	30
Intermediate population size	80	78	80	10
Replace generational	true	false	false	False
Replace only if better	false	false	true	False
Total work	220000	214500	220000	214500

Tuning of other parameters. There are some other parameters whose values were to be fixed. These are: (a) `independent_runs` –indicates the number of independent runs to perform; This parameter is useful for obtaining computational results of statistical relevance. Its meaning is that the GA implementation is run that many times over the same instance using the same configuration; (b) `nb_iterations` –indicates the number of iterations within an independent run. In the GA implementation it is equal to the number of evolution steps; and (c) `max_time_to_spend` (in seconds) –indicates the maximum execution time within an independent run. Note that the GA skeleton offers the possibility to implement the stopping condition using the last two parameters so that GA terminates when either the number of iterations or the maximum time is reached. The `independent_runs` is fixed to 10 (reported results are then averaged) and `max_time_to_spend` is fixed to 90 secs (the execution time in [5] is in average 60 secs).

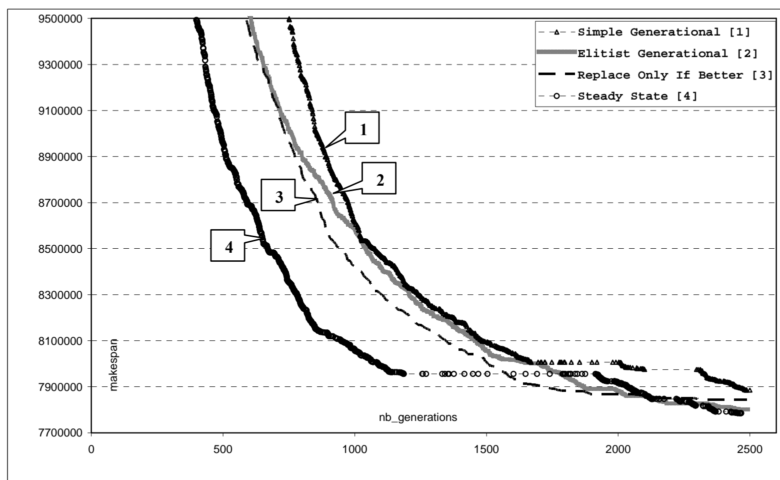

FIGURE 7. Comparison of different replacement operators

4.2. **Computational results for static instances.** We summarize here some of the computational results obtained with GA implementation for instances from [5].

The simulation model in [5] is based in ETC (Expected Time To Compute) matrix. The instances of the benchmark in [5] are classified into 12 different types of ETC matrices according to three metrics: job heterogeneity, machine heterogeneity and consistency. An ETC matrix is considered consistent when, if a machine m_i executes job t 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. All instances consist of 512 jobs and 16 machines and are labelled $u_x_yyzz.0$ whose meaning is the following:

- u means uniform distribution (used in generating the matrix).
- x means the type of inconsistency (c–consistent, i–inconsistent and s means semi-consistent).
- yy indicates the heterogeneity of the jobs (*hi*–high, and *lo*–low).
- zz indicates the heterogeneity of the resources (*hi*–high, and *lo*–low).

A total of 12 instances, which are used in the reference work are used for purposes of the computational results. These instances consist in fact of three groups of four instances each: the first group represents consistent ETC matrices, the second one represents inconsistent matrices and the third one represents semi-consistent matrices. For each group, instances *hihi*, *hilo*, *lohi* and *lolo* are considered. In this way, the set of considered instances covers the main characteristics of distributed heterogeneous computing environments.

We recall that GA implementations described in this paper optimize makespan and flowtime (in both hierarchic and simultaneous approach) while GA implementation of Braun et al. [5] minimizes only the makespan, therefore we will first compare the results just for the makespan. Note however that optimizing also the flowtime had an additional computational cost, which nonetheless doesn't seem to affect the quality of solutions found by our GA implementations.

TABLE 5. Comparison of the makespan values for static instances

Instance	Min-Min	MCT LJFR-SJFR	GA Braun Best makespan	GA hierarchic Best makespan	GA simultaneous Best makespan
u_c_hihi.0	8460675.000	14665600.200	8050844.500	7730973.882	**7610176.437**
u_c_hilo.0	164022.440	213423.330	156249.200	157255.844	**155251.196**
u_c_lohi.0	275837.340	485591.120	258756.770	252907.580	**248466.775**
u_c_lolo.0	5546.260	7112.790	5272.250	5310.615	**5226.972**
u_i_hihi.0	3513919.250	4193476.360	3104762.500	3141395.822	**3077705.818**
u_i_hilo.0	80755.680	92003.300	**75816.130**	77455.085	75924.023
u_i_lohi.0	120517.710	145157.280	107500.720	108047.402	**106069.101**
u_i_lolo.0	2779.090	3296.480	2614.390	2676.207	**2613.110**
u_s_hihi.0	5160343.000	6510165.670	4566206.000	4745284.938	**4359312.628**
u_s_hilo.0	104540.730	121170.490	98519.400	101817.924	**98334.640**
u_s_lohi.0	140284.480	190442.120	130616.530	137785.504	**127641.889**
u_s_lolo.0	3867.490	4438.420	3583.440	3650.316	**3515.526**

Braun et al.'s GA configuration. It is worth noting the configuration parameters of the GA implementation of [5]. Crossover operator: *CrossOnePoint* ($p_c = 0.6$); Mutation operator: *MutateMove* ($p_m = 0.4$); Selection operator: *Linear Ranking* and Replacement operator: *Elitist Generational*. One important aspect of GA implementation of [5] is the use of Min-Min heuristic for generating initial solutions, which is shown to achieve a very good reduction in the makespan. A population of size 200 is used. Finally, their GA finalizes when either 1000 iterations (number of evolution steps) have been executed or when the chromosome elite has not varied during 150 iterations.

In order to compare the results of their and our GA implementations, we computed the work[2] W made by both implementations using their respective configurations. Thus, in Braun et al. GA, $W = (0.6 + 0.4) * 200 * 1000 = 200000$ and in our implementation $W = (0.8 + 0.4) * 68 * 2500 = 204000$, from which we see that a comparable amount of work is done, assuring a fair comparison.

Finally, Braun et al. used a Pentium III 400Mhz, 1Gb RAM and we used an AMD K6(tm) 3D 450 MHz and 256 Mb of RAM under Linux/Debian OS.

Results for makespan parameter. We give in Table 5 the computational results for makespan[3]. In the table, the first column indicates the instance name, the second column indicates the makespan of the initial solution obtained with Min-Min, the third column indicates the best makespan of the initial solutions found by MCT and LJFR-SJFR, the fourth, fifth and sixth columns indicate the best makespan found by the Braun's GA, our GA (hierarchic version) and our GA (simultaneous version), respectively. Both GAs (hierarchic and simultaneous version) were initialized using MCT, LJFR-SJFR and Random.

[2]Recall that $W = (p_c + p_m) * pop_size * nb_evolution_steps$.

[3]Results are obtained by running GAs on the same instance and under the same configuration 10 timres.

From Table 5 we observe that our implementation outperforms the results of [5] for all but one instance, the *u_i_hilo*.0 (inconsistent matrix with low heterogeneity of resources and high heterogeneity of jobs). It is worth mentioning here that our initial populations could be worse than the ones used in [5] due to the initializing heuristics used (Min-Min performs much better than LJFR/SJFR and MCT). In fact, by monitoring some of the runs, we observed that our GA spends roughly 55-70% of the total number of iterations to reach a solution of the quality of Min-Min. We would thus expect an even better performance of our GA if initialization was conducted using Min-Min. On the other hand our version of hierarchic optimization obtains results similar to those of [5] but does not outperform it (except two instances).

Notice that the GA with hierarchic optimization criteria, obtains better results only for instances with consistent ETC matrices having a high level of heterogeneity of resources. For the rest of instances the deviation with respect to the Braun et al. GA is 2.9% in average (5.91% in the worst case). The standard deviation of the average makespan of our GA implementations from the best found makespan are less than 1% implying that our implementations are robust, in that they produce very good solutions for all reported executions.

Results for flowtime parameter. Computational results for flowtime value are given in Table 6 using the simultaneous approach, which performed better than the hierarchic one. Two versions of the implemented GAs that yielded better results are compared. As can be seen from the table, the Steady Sate GA performs better for consistent and semi-consistent matrices and Replace If Better GA performs better for inconsistent matrices.

TABLE 6. Flowtime values obtained with two GA versions

Instance	GA (Replace If Better)	GA (Steady State)
u_c_hihi.0	1073774996	**1048333229**
u_c_hilo.0	28314677.9	**27687019.4**
u_c_lohi.0	35677170.8	**34767197.1**
u_c_lolo.0	942076.61	**920475.17**
u_i_hihi.0	**376800875**	378010732
u_i_hilo.0	12902561.3	**12775104.7**
u_i_lohi.0	**13108727.9**	13444708.3
u_i_lolo.0	**453399.32**	446695.83
u_s_hihi.0	541570911	**526866515**
u_s_hilo.0	17007775.2	**16598635.5**
u_s_lohi.0	15992229.8	**15644101.3**
u_s_lolo.0	616542.78	**605375.38**

5. **Conclusions and future work.** We presented an extensive study on the usefulness of Genetic Algorithms (GAs) for designing efficient Grid schedulers when makespan and flowtime parameters are minimized under hierarchic and simultaneous approaches. Rather than a single implementation, a family of GAs are illustrated with two encodings and different types of operators for each of them. Most important conclusions of this work are:

- The experimental study reveals the quality of the proposed GA-based schedulers as compared well to the existing GA-schedulers in the literature.
- Our experimental study shows the performance of different GA operators.
- Our GA-based schedulers can be used to design dynamic schedulers. A dynamic scheduler would run our GA in batch mode to schedule jobs arrived in the system since last activation of the scheduler.
- Our GA-based schedulers can be run adaptively if we new in advance grid characteristics such as consistency of computing, heterogeneity of resources and jobs.

Our future work is targeted on dynamic scheduling.

Acknowledgement. Research is partially supported by Projects ASCE TIN2005-09198-C02-02, FP6-2004-IST-FETPI (AEOLUS) and MEC TIN2005-25859-E. The authors also gratefully acknowledge the helpful comments of the reviewers.

REFERENCES

[1] A. Abraham, R. Buyya, and B. Nath. Nature's heuristics for scheduling jobs on computational grids. In *The 8th IEEE International Conference on Advanced Computing and Communications (ADCOM 2000)*, India, 2000.

[2] A. Abraham, H. Liu, W. Zhang and T.G. Chang, Job Scheduling on Computational Grids Using Fuzzy Particle Swarm Algorithm, *10th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, B. Gabrys et al. (Eds.): Part II, Lecture Notes on Artificial Intelligence 4252, 500507, Springer, 2006.

[3] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems Journal*, 18(8):1061–1074, 2002.

[4] E. Alba, F. Almeida, M. Blesa, C. Cotta, M. Daz, I. Dorta, J. Gabarr, C. Len, G. Luque, J. Petit, C. Rodrguez, A. Rojas, and F. Xhafa. Efficient parallel LAN/WAN algorithms for optimization. the mallba project. *Parallel Computing*, 32(5-6):415–440, 2006.

[5] H.J. Braun, T. D.and Siegel, N. Beck, L.L. Blni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, and B. Yao. 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.

[6] R. Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, Melbourne, Australia, 2002.

[7] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *The 4th International Conference on High Performance Computing*, Asia-Pacific Region, China, 2000.

[8] J. Cao, D.P. Spooner, S.A. Jarvis and G.R. Nudd, Grid load balancing using intelligent agents, *Future Generation Computer Systems*, 21(1), 135-149, 2005.

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

[10] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The apples parameter sweep template: user-level middleware for the grid. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, page 60. IEEE Computer Society, 2000.

[11] I. Foster and C. Kesselman. *The Grid - Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[12] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *International Journal of Supercomputer Applications*, 15(3), 2001.

[13] M.R. Garey and D.S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[14] Y. Gao, H. Rong and J.Z. Huang, Adaptive grid job scheduling with genetic algorithms, *Future Generation Computer Systems*, 21(1), 151-161, 2005,

[15] T. Gruninger. Multimodal optimization using genetic algorithms. Master's thesis, Stuttgart University, 1996.

[16] S. Hotovy. Workload evolution on the Cornell theory center IBM SP2. In *Job Scheduling Strategies for Parallel Processing Workshop, IPPS'96*, 27–40, 1996.

[17] K.P. Kumar, A. Agarwal, and R. Krishnan. Fuzzy based resource management framework for high throughput computing. In Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid, 555–562, 2004. IEEE Computer Society.

[18] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999.

[19] V. Di Martino and M. Mililotti. Sub optimal scheduling in a grid using genetic algorithms. *Parallel Computing*, 30:553–565, 2004.

[20] J. Page and J. Naughton. Framework for task scheduling in heterogeneous distributed computing using genetic algorithms. *Artificial Intelligence Review*, 24:415–429, 2005.

[21] G. Ritchie. Static multi-processor scheduling with ant colony optimisation & local search. Master's thesis, School of Informatics, University of Edinburgh, 2003.

[22] G. Ritchie and J. Levine. A fast, effective local search for scheduling independent jobs in heterogeneous computing environments. Technical report, Centre for Intelligent Systems and their Applications, School of Informatics, University of Edinburgh, 2003.

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

[24] D. Whitley. The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best. In D. Schaffer, ed., *Proceedings of the 3rd International Conference on Genetic Algorithms*, 116–121. Morgan Kaufmann, 1989.

[25] A. Yarkhan and J. Dongarra. Experiments with scheduling using simulated annealing in a grid environment. In *3rd International Workshop on Grid Computing (GRID2002)*, 232–242, 2002.

[26] A.Y. Zomaya and Y.H. Teh. Observations on using genetic algorithms for dynamic load-balancing. *IEEE Transactions On Parallel and Distributed Systems*, 12(9):899–911, 2001.