Computing and Informatics, Vol. 28, 2009, 237–250

A TABU SEARCH ALGORITHM FOR SCHEDULING INDEPENDENT JOBS IN COMPUTATIONAL GRIDS

Fatos Xhafa, Javier Carretero

Department of Languages and Informatics Systems
Polytechnic University of Catalonia, Spain
e-mail: fatos@lsi.upc.edu

Bernabé Dorronsoro

Faculty of Science, Technology and Communication
University of Luxembourg, Luxembourg
e-mail: bernabe.dorronsoro@uni.lu

Enrique Alba

Deptartment of Languages and Computer Science
University of Málaga, Spain
e-mail: eat@lcc.uma.es

Revised manuscript received 9 December 2008

Abstract. The efficient allocation of jobs to grid resources is indispensable for high performance grid-based applications, and it is a computationally hard problem even when there are no dependencies among jobs. We present in this paper a new tabu search (TS) algorithm for the problem of batch job scheduling on computational grids. We define it as a bi-objective optimization problem, consisting of the minimization of the makespan and flowtime. Our TS is validated versus three other algorithms in the literature for a classical benchmark. We additionally consider some more realistic benchmarks with larger size instances in static and dynamic environments. We show that our TS clearly outperforms the compared algorithms.

Keywords: Job scheduling, computational grid, tabu search

Mathematics Subject Classification 2000: 68T20

1 INTRODUCTION

Computational Grid (CG) is a new distributed computing paradigm for the development of large-scale distributed applications [4]. CGs currently represent a very successful approach for large-scale distributed real-world applications [14]. Nonetheless, the grid computing paradigm is still raising important issues to be solved, like the efficient dynamic allocation of jobs to geographically distributed grid resources.

Although the family of scheduling problems is one of the most studied ones by the optimization research community, the application of the available approaches to the job scheduling on CGs is not straightforward, as it differs significantly from conventional scheduling in distributed systems. The reason is that scheduling in grid systems adds new features not present in conventional scheduling problems. Some examples are the heterogeneity of jobs, multi-objectivity (several – possibly in conflict – objectives to reach, e.g., makespan, flowtime, or resource utilization), the intrinsic dynamic nature of grids (jobs that are arriving, new resources arriving or leaving, ...), or the heterogeneity of resources and networks.

In this work, we are considering the scheduling of independent jobs, one simple yet very important version of the problem. Given the large size and the decentralized nature of grids, this type of scheduling arises naturally when independent users or applications are continuously submitting jobs to the grid. One important implication of this is that any grid scheduler must achieve allocations of jobs to resources in very short times and must be able to adapt itself to the changes of the grid.

Heuristic methods have turned out to be a standard approach in combinatorial optimization. Dealing in practice with real size problems makes the use of such methods the *de facto* choice. One such method is the Tabu Search (TS), which has shown its effectiveness in a broad range of combinatorial optimization problems. In this work, we propose a new TS algorithm for a bi-objective definition of the job scheduling problem on computational grids, consisting of the minimization of makespan and flowtime.

The TS algorithm distinguishes for its flexibility in exploiting domain/problem knowledge in the selection of parameters and other inner components. The TS implementation presented here explores this flexibility, and thus by carefully designing and implementing its sub-algorithms and tuning the parameters, our implementation is able to outperform other known heuristic approaches for a well-accepted benchmark. We completed our experiments by analyzing the behavior of our TS on more realistic benchmarks than the previous one, composed of larger size instances (having up to 256 machines) in both static and dynamic environments.

The paper is organized as follows. In Section 2 we give the description of job scheduling problem in CGs considered in this work. TS and its particularization for the problem are given in Section 3. Next, we present our experimental study

in Section 4. In Section 5 we summarize our most important results, and indicate directions for further work.

2 PROBLEM DEFINITION

In this section we present the job scheduling problem in computational grids. In this paper we are using the model of Braun et al. [2] for simulating heterogeneous distributed environments. Using this model will allow us to make realistic simulations of the grid system. It is presented in Section 2.1. Then, in Section 2.2 we describe the bi-objective optimization problem we define for solving the problem.

2.1 The ETC Model

In the ETC model [2], a collection of independent jobs is considered for allocation of resources. The model is based on the definition of the Expected Time to Compute (ETC) matrix, in which $ETC[j][m]$ indicates an estimation of how long will it take to complete job j on resource m. One possible way to compute the $ETC[j][m]$ values is to divide the workload of job j by the computing capacity of resource m. We are assuming here that the workload is known and, in practice, it can be obtained from specifications provided by the user, from historical data, or from predictions [7, 8].

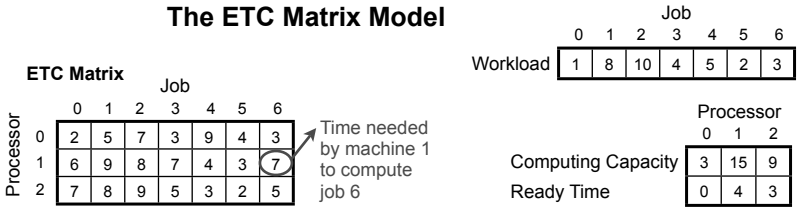


Fig. 1. The ETC matrix model is composed by the jobs workload, the processors computing capacity and ready times, and the time required by processors to finish the jobs

Using the ETC matrix model, an instance of the job scheduling, at an instant of time, can be defined as (see Figure 1) a number of independent *jobs* to be allocated to grid resources, a number of possible *machines* for the allocation of jobs, the *workload* (in millions of instructions) of each job, the *computing capacity* of each machine (in *mips*), the ready times, denoted $ready_m$, indicating when machine m can start computing the assigned jobs, and the ETC matrix (with size $nb\_jobs \times nb\_machines$), where $ETC[j][m]$ is the value of the expected time to compute of job j in machine m.

2.2 Objective Function

The problem of scheduling jobs for computational grids is in fact a multiobjective task, since the we can use several criteria for measuring the quality of solutions.

Some examples are the *makespan* (the finishing time of the latest job), the *flowtime* (the sum of finalization times of all the jobs), the *completion time* of jobs in every machine (closely related to makespan), or maximizing the resource utilization.

In this work we are considering the minimization of *makespan* and *flowtime* defined in (1), where F_j denotes the time when job j finalizes and $Sched$ is the set of all possible schedules in a single weighted function. On the one hand, makespan is an indicator of the general productivity of the grid system (it is very commonly used in scheduling problems). On the other hand, flowtime is an indicator of the response time to the user petitions for job executions. Small response times indicate high QoS of the system.

$$makespan = \min_{S_i \in Sched} \left\{ \max_{j \in Jobs} F_j \right\} \; ; \; flowtime = \min_{S_i \in Sched} \left\{ \sum_{j \in Jobs} F_j \right\} . \quad (1)$$

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 an ascending order of their ETC value. In fact, makespan and flowtime are contradictory objectives, in the sense that trying to minimize one of them could be in detriment of the other, especially for near-optimal schedules.

For solving the problem, we designed a hierarchical algorithm in which the two objectives are optimized in different steps: the algorithm first optimizes the considered most important objective (makespan) and after that it optimizes the secondary goal, namely, the flowtime, without worsening the makespan value. The objective value of a given solution S is then computed as (being λ a predefined parameter for weighting the importance of every objective):

$$fitness(S) = \lambda \cdot makespan(S) + (1 - \lambda) \cdot \frac{flowtime(S)}{nb\_machines}. \quad (2)$$

3 DESCRIPTION OF THE TABU SEARCH ALGORITHM

We present here the TS algorithm proposed for solving the problem described in Section 2. TS was introduced as a high-level algorithm that uses other specific heuristics to guide the search [5]; the goal is performing an intelligent exploration of the search space that would allow to avoid getting trapped into local optima. In Algorithm 1 we show a generic pseudocode for TS. This general template offers many different possibilities for highly specializing our TS algorithm for a concrete problem just by designing its inner heuristics and appropriate data structures.

As can be seen from the template of Algorithm 1, one of the distinguishing features of TS versus other heuristics is the use of a *historical memory*, which consists of a *short term memory* (or *recency*) with information on recently visited solutions, and a *long term memory* (or *frequency*) storing information gathered during the

Algorithm 1 A Template for Tabu Search Algorithm

```
Compute an initial solution s; let ŝ ← s;
Reset the tabu and aspiration conditions;
while not termination-condition do
  Generate a subset of solutions N*(s) that do not violate the tabu conditions or hold
  the aspiration criteria;
  Choose the best s' ∈ N*(s) with respect to the cost function; s ← s';
  if improvement(s', ŝ)) then ŝ ← s';
  Update the recency and frequency;
  if (intensification condition) then Perform intensification procedure;
  if (diversification condition) then Perform diversification procedures;
end while
return ŝ;
```

whole complete exploration process. The movements in these two lists are considered tabu and then they cannot be used. Thus, some *aspiration criteria* are needed for removing the tabu movements. Additionally, we need to specify some inner heuristics like the *local search*, used for exploring the neighborhood of a solution, or the *intensification and diversification procedures*, for appropriately managing the exploration/exploitation tradeoff on the search space. We describe the components of our algorithm next:

Solution representation. A schedule is represented as a vector of size *nb_jobs*, in which *schedule*$[i]$ indicates the machine where job i is assigned to.

Movements. Two types of movements are considered for this representation [13]: *transfer* and *swap*. Transfer moves a job from one machine to another, while swap exchanges two jobs assigned to different machines.

Initial solution. It is generated with the *Min-Min* method. It starts by computing a matrix *completion*$[i][j]$ with the time when every job would finish in each machine (*completion*$[i][j]$ = *ETC*$[i][j]$ + *ready*$[j]$). Then we choose the job k and machine m with the earliest completion time (*completion*$[k][m] \leq$ *completion*$[i][j]$ $\forall i, j$), remove k from the set of jobs, and update *completion*$[i][j]$ for the rest of jobs. This process is repeated until all jobs are assigned.

Historical memory. Three different memories have been used: 1) the *recency* memory (a tabu list with the last time each job was assigned to every machine [12]), 2) a tabu hash table with the visited solutions, and 3) a *frequency* memory (storing how many times every job has been assigned each machine).

Aspiration criteria. Two criteria are used to accept tabu movements: 1) fitness (they are accepted when yielding to better solutions), and 2) makespan (movements improving the makespan of the solution are allowed).

Neighborhood exploration. It is done by means of load balancing: movements made among jobs assigned to most- and less-loaded machines. Our TS considers first the neighbors providing the best improvements – in terms of the completion

time, i.e., the time when a machine finishes all its tasks, see (3) – in the solution, or the least worst movement in case no better solutions are found.

$$completion[m] = ready\_times[m] + \sum_{\{j \in Jobs \mid schedule[j]=m\}} ETC[j][m] \ , \qquad (3)$$

Intensification. It is executed when there is evidence that the current solution is in a promising region. Thus, a deeper exploration of this area is done by rewarding (attributes of) the current solution, forcing their presence in the new ones. For that we use a combination of the following three strategies (they are applied in this order one by one until no improvements are possible):

1. The most promising attributes of solutions (from the frequency memory) are rewarded. The most frequent job assignments are chosen with probability .75, while in other case (probability .25) a roulette-wheel is used in the assignment.
2. The values for the maximum and minimum load factor parameters are temporarily changed by using the current state of the grid (in terms of the workload of machines) with the only condition that the number of resulting transfers and swaps are bounded by their respective upper bounds. Then, the neighborhood is defined by these new values.
3. The structure of the neighborhood is changed by applying all the possible swap movements between two machines, and then performing just one transfer from one machine to the other.

Diversification. We use three *soft* diversification methods (performing slight perturbations to the solution), and a *strong* diversification (hard perturbation):

- *Using the job distribution* (or *influential diversification* [6]). It lies in redistributing the jobs to machines so that both long and short jobs are assigned to every machine, thus improving the load balancing of resources.
- *Using penalization of ETC values.* We use the *frequency* memory to penalize the corresponding ETC values of most frequent job-machine assignments.
- *Freezing jobs.* Jobs that have most frequently changed their assignments are frozen (we set tabu to all movements involving the job during the iteration). After the diversification, the tabu status of this (these) job(s) is cancelled.
- *Strong diversification.* We perform a large perturbation of the current solution by randomly changing the assignments of a sufficient number of jobs.

4 EXPERIMENTAL STUDY

In this section, we present the results we obtained with our TS implementation. Our tests were made on a Pentium III 550MHz, 256MB RAM, and the algorithm was implemented in C++ using the MALLBA framework [1]. We validate our TS by

comparing it with some other algorithms in the literature on a classical benchmark of the problem (Section 4.1). After that, we tackle two new benchmarks composed by a set of much larger static (Section 4.2) and dynamic (Section 4.3) instances.

4.1 Validation of the Algorithm in a Classical Benchmark

The objective of the study presented in this section is to show the quality of the TS algorithm we propose in this work. For that, our algorithm is compared versus some other optimization algorithms that were previously applied in the literature to the considered problem. The benchmark of instances considered in this study is presented in Section 4.1.1, while the parametrization used for the TS algorithm for solving these problems is given in Section 4.1.2. Finally, our results and the comparison with other algorithms in the literature is provided in Section 4.1.3.

4.1.1 Benchmark Description

This section describes the benchmark by Braun et al. [2]. It is a frequently used benchmark, very effective in simulating grid systems and capturing most important characteristics of the job scheduling problem. In it, instances are classified according to three parameters (job heterogeneity, machine heterogeneity, and consistency) into 12 different types of *ETC* matrices, each of them consisting of 100 instances. All instances are composed from 512 jobs and 16 machines. They are labelled as $u\_x\_yyzz.k$ where u means uniform distribution (in the matrix generation), x is the type of consistency (c – consistent, i – inconsistent and s means semi-consistent), yy and zz indicate the job and machine heterogeneity (hi – high, and lo – low), and k is used to number instances of the same type. An ETC matrix is consistent when if a machine is faster than other for some job, then it is faster for all the jobs. Inconsistency means that a machine is faster for some jobs and slower for some others, while it is semi-consistent if it contains a consistent sub-matrix.

4.1.2 Parametrization

In this section we present the parameters of our algorithm. The size of the tabu hash table (TH) is set to 918 133, which is a high number and non divisible by 20, as it is recommended by Srivastava [11]. The maximum number of iterations a solution remains tabu (max_tabu_status) is chosen uniformly from the interval $[nb\_machines, 2 \cdot nb\_machines]$, and the maximum number of successive iterations without improvements of the current solution implying the activation of the intensification is fixed to $4\ln(nb\_jobs)\cdot\ln(nb\_machines)$. The number of iterations of a diversification and an intensification are set to $\log_2(nb\_jobs)$. Finally, we consider a number of 30 elite solutions, and a value of $(max\_tabu\_status/2) - \log_2(max\_tabu\_status)$ for the minimum number of iterations after which a tabu movement can aspire. The algorithm runs for 100 seconds (a realistic run time for scheduling tasks in grids).

Additionally, we did not use a fast computer in our tests, so we are providing here lower bounds for the results, that are expected to be improved with faster machines.

4.1.3 Computational Results

Here we compare the proposed TS algorithm to other state-of-the-art algorithms for the classical benchmark described in Section 4.1.1. These compared algorithms are a TS implementation and a hybrid ACO+TS algorithm due to Ritchie [10], and a cellular memetic algorithm cMA hybridized with a TS as local search step [15].

The four algorithms are compared in Table 1 in terms of the average makespan values. We can not compare them in terms of flowtime because the authors of the other algorithms do not provide this result. In the case of both our TS algorithm and the cMA, each reported value is the average makespan value out of 10 executions. Conversely, in the case of the other two compared algorithms, the results were obtained after one single run, as it is reported in the original work.

As can be seen in Table 1, our TS outperforms the other algorithms for 8 out of the 12 considered instances (see the **bold** values). In the other 4 instances, ACO+TS provides the best results, but differences with respect to our TS algorithm (which is the second best one for these instances) are small. Additionally, we should mention at this point the reduced execution time of 100 seconds fixed as the stopping criterion for our TS, which is far lower than Ritchie's run times (over 3.5 hours).

In order to measure the accuracy of our proposal, we computed the deviation of the average makespan with respect to our best makespan value (in percentage), and the average value obtained for this parameter in all the tested instances is 0.112 %.

Instance	TS Ritche et al.	ACO+TS Ritche et al.	cMA Xhafa et al.	Our TS
u_c_hihi.0	7 568 871.83	7 497 200.85	7 554 119.35	**7 458 864**.45
u_c_hilo.0	154 644.48	154 234.63	154 057.58	**153 438**.08
u_c_lohi.0	245 981.55	244 097.28	247 421.28	**242 385**.38
u_c_lolo.0	5 202.51	5 178.44	5 184.79	**5 155**.78
u_i_hihi.0	3 021 155.10	**2 947 754**.12	3 054 137.65	2 959 029.35
u_i_hilo.0	74 400.68	73 776.24	75 005.49	**73 734**.84
u_i_lohi.0	104 309.12	**102 445**.82	106 158.73	103 867.13
u_i_lolo.0	2 580.62	**2 553**.54	2 597.02	2 559.96
u_s_hihi.0	4 248 200.21	**4 162 547**.92	4 337 494.59	4 181 985.83
u_s_hilo.0	97 711.72	96 762.00	97 426.21	**96 432**.14
u_s_lohi.0	126 115.39	123 922.03	128 216.07	**123 600**.51
u_s_lolo.0	3 505.69	3 455.22	3 488.30	**3 454**.02

Table 1. Comparison versus other algorithms in the literature. Average makespan values.

Summarizing, the proposed TS outperforms the compared algorithms for all the instances. The exceptions are the inconsistent ones, for which the ACO+TS obtains slightly better results. We believe that this better performance of our TS implementation is due to a better embedding of problem specific knowledge into the

components of the algorithm. Additionally, our TS algorithm runs for 100 seconds, much less than the 3.5 hours of ACO+TS. This is an important result, since the execution time of the scheduler in the dynamic environment of grid systems is a critical factor. Thus, the short execution times achieved by our TS yields to fast and significant reductions of makespan, making it very suitable for real grid schedulers.

4.2 Experimentation with Larger Size Static Instances

After validating our TS algorithm in Section 4.1 with other existing algorithms in the literature, we proceed now to analyze its behavior with more realistic problems. For that, we generated a new benchmark by extending the *HyperSim* open source package [9] to simulate a grid system. The parametrization used for the simulator (shown in Table 2) has been carefully set in order to have different kinds of real grids. This way, we have defined grids of different sizes (all of them larger than those of Section 4.1.1), grouped into four different sets called small (32 hosts/512 tasks), medium (64 hosts/1 024 tasks), large (128 hosts/2 048 tasks), and very large (256 hosts/4 096 tasks). Both the capacity of resources and the workload of tasks are randomly set following a normal distribution. Finally, all resources of the system can be used, all tasks must be scheduled, and the benchmark is created from the average results obtained after 15 runs of the simulator.

	Small	**Medium**	**Large**	**Very Large**
Number of hosts	32	64	128	256
Resource capacities (in MIPS)	$N(1\,000, 175)^*$			
Total number of tasks	512	1 024	2 048	4 096
Workload of tasks	$N(2.5 * 10^8, 4.375 * 10^7)$			
Host selection	All			
Task selection	All			
Number of runs	15			

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

Table 2. Settings for the grid simulator for generating large static instances

In Table 3 we give the makespan and flowtime values obtained by our TS algorithm for the new larger set of instances. In Table 3 we observe that makespan value increases slowly as the instance size is doubled, while flowtime increases considerably. It is almost doubled when doubling the instance size.

Instance size	**Small**	**Medium**	**Large**	**Very large**
Makespan ±% C.I (0.95)	3 969 016.842	3 970 894.032	3 980 381.381	3 972 429.011
	±0.3833 %	±0.4103 %	±0.4092 %	±0.4074 %
Flowtime ±% C.I (0.95)	104 786 744.1	210 295 284.0	419 926 173.3	833 351 728.9
	±0.9284 %	±0.8355 %	±0.8440 %	±0.9033 %

Table 3. Makespan valued for larger size instances (C.I. – Confidence Interval)

4.3 Experimentation with Dynamic Instances

As an extension to the experiments carried out in previous sections, we proceed here to apply our TS algorithm to a more realistic benchmark composed by dynamic instances (from 32 to 256 machines) in order to evaluate its performance in more realistic scenarios. The application of the scheduler to dynamic instances is possible because its high speed finding quality planning allows us to run it in batch mode. In this case we use the same simulator, but with some additional parameters (explained below). The TS-based scheduler is plugged into the Grid simulator. When an event `schedule` occurs in the simulator, the problem instance is passed to the TS scheduler to find a good scheduling. This solution of the TS is then sent back to the simulator to generate the new state of the grid. This process is repeated until all jobs are scheduled. Results are averaged over 15 runs of the simulator.

	Small	**Medium**	**Large**	**Very Large**
Init. hosts [max., min.]	32 [37, 27]	64 [70, 58]	128 [135, 121]	256 [264, 148]
MIPS	$N(1\,000, 175)$			
Add host	$N(625\,000, 93\,750)$	$N(562\,500, 84\,375)$	$N(500\,000, 75\,000)$	$N(437\,500, 65\,625)$
Delete host	$N(625\,000, 93\,750)$			
Total tasks	512	1 024	2 048	4 096
Init. tasks	384	768	1 536	3 072
Workload	$N(2.5 * 10^8, 4.375 * 10^7)$			
Interarrival	$E(7\,812.5)^\dagger$	$E(3\,906.25)$	$E(1\,953.125)$	$E(976.5625)$
Activation	Resource_and_time_interval (250 000)			
Reschedule	True			
Host select	All			
Task select	All			
Number of runs	15			

$^\dagger E(\mu)$ is an exponential distribution an average value μ

Table 4. Settings for the dynamic grid simulator

The parameters used in this case are shown in Table 4. As can be seen, the value for the number of hosts is now defined as an initial value and an interval in which this value can vary during the simulation, emulating resources that become available and unavailable in the grid (the frequency of appearing and disappearing resources is defined with the normal distributions given by *add host* and *delete host*, respectively). The initial number of tasks is given by *init. tasks*, and new tasks arrive with frequency *interarrival* until *total tasks* is reached. *Activation* establishes the activation policy according to an exponential distribution. The already scheduled tasks that have not been executed yet will be rescheduled if *reschedule* is true.

In Table 5 we give the makespan and flowtime values obtained for this dynamic benchmark. We observe here a similar behavior as in the case of the large static instances (see Section 4.2): the makespan value increases slowly as the instance size is doubled, while the flowtime value is also doubled. Additionally, in Table 5 we compare the performance of our TS versus a steady-state genetic algorithm (ssGA) proposed by Carretero and Xhafa in [3]. The algorithms are compared only in terms of the makespan because no values for flowtime were reported in the referred work. As can be seen, our TS outperforms the compared algorithm for the studied instances, as it previously happened in Section 4.1 for the static benchmark. Thus,

our TS is a robust solution that outperformed some of the state-of-the-art algorithms both in static and dynamic environments.

Size	Makespan ± % C.I (0.95)		Flowtime ± % C.I (0.95)
	TS	**ssGA**	**TS**
32	$\mathbf{3\,969\,016.8}_{\pm 0.4\%}$	$4\,063\,425.5_{\pm 0.8\%}$	$104\,786\,744.1_{\pm 0.9\%}$
64	$\mathbf{3\,970\,894.0}_{\pm 0.4\%}$	$3\,994\,804.9_{\pm 0.9\%}$	$210\,295\,284.0_{\pm 0.8\%}$
128	$\mathbf{3\,980\,381.4}_{\pm 0.4\%}$	$3\,995\,162.0_{\pm 1.3\%}$	$419\,926\,173.3_{\pm 0.8\%}$
256	$\mathbf{3\,972\,429.0}_{\pm 0.4\%}$	$4\,009\,852.0_{\pm 1.8\%}$	$833\,351\,728.9_{\pm 0.9\%}$

Table 5. Makespan and flowtime values for dynamic instances (C.I.: Confidence Interval)

5 CONCLUSIONS AND FURTHER WORK

In this work we have presented a Tabu Search (TS) implementation for scheduling independent jobs in grid systems. This scheduling problem is currently receiving considerable attention from researchers due to its importance in obtaining high performance applications for running highly costly algorithms using grid systems. TS has been considered here to cope with the complexity of the problem and because it has shown to be very effective for a variety of optimization problems, including scheduling problems. As a matter of fact, TS has been previously considered for solving the scheduling problem by Ritchie and Levine in 2004, but the reported execution times are prohibitive for a grid system given its dynamic nature. Therefore, our main objective was to obtain an efficient implementation that would yield to a scheduler for realistic grid systems. Our computational results show that our TS scheduler outperforms Ritchie's implementations for most of the considered instances at far inferior executions times. Additionally, our TS has also been tested in more realistic frameworks (larger static and dynamic instances), outperforming also previous approaches.

In our further work we would like to go deeper into the experimental study of the TS scheduler in the dynamic setting. Also, we would like to consider much larger size instances, and to address parallelization of the TS implementation using existing parallel models for the method in the literature. Finally, another interesting issue is the decentralized definition of the problem, so that the grid can support more than one scheduler.

Acknowledgment

The research was partially supported by ASCE TIN2005-09198-C02-02, FP6-2004-IST-FETPI (AEOLUS) and MEC TIN2005-25859-E Projects. E. Alba acknowledges partial support for project P07-TIC-03044 (`http://diricom.lcc.uma.es/`).

REFERENCES

[1] Alba, E.—Almeida, F.—Blesa, M.—Cotta, C.—Díaz, M.—Dorta, I.—Gabarró, J.—León, C.—Luque, G.—Petit, J.—Rodríguez, C.—Rojas, A.—Xhafa, F.: Efficient parallel LAN/WAN algorithms for optimization. The Mallba project. Parallel Computing, Vol. 32, 2006, No. 5–6, pp. 415–440.

[2] Ali, S.—Siegel, H.J.—Maheswaran, M.—Hensgen, D.—Ali, S.: Representing Task and Machine Heterogeneities for Heterogeneous Computing Systems. Tamkang Journal of Science and Engineering, Vol. 3, 2000, No. 3, pp. 195–207.

[3] 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 Univ., Vol. 12, 2006, No. 1, pp. 11–17.

[4] Foster, I.—Kesselman, C.: The Grid – Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, 1998.

[5] Glover, F.: Future Paths for Integer Programming and Links to Artificial Intelligence. Computers and Op. Res., Vol. 5, 1986, pp. 533–549.

[6] Hübscher, R.—Glover, F.: Applying Tabu Search With Influential Diversification to Multiprocessor Scheduling. Comput. Oper. Res., Vol. 21, 1994, No. 8, pp. 877–884.

[7] Hotovy, S.: Workload Evolution on the Cornell Theory Center IBM SP2. In Job Scheduling Strategies for Parallel Proc. Workshop, IPPS '96, 1996, pp. 27–40.

[8] The Hebrew University Parallel Systems Lab. Parallel workload archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`.

[9] Phatanapherom, S.—Kachitvichyanukul, V.: Fast Simulation Model for Grid Scheduling Using Hypersim. In Winter Simulation Conf., Vol. 2, 2003, pp. 1494–1500.

[10] Ritchie, G.—Levine, J.: A Hybrid Ant Algorithm for Scheduling Independent Jobs in Heterogeneous Computing Environments. In Workshop of the UK Planning and Scheduling Special Interest Group, PLANSIG 2004, 2004.

[11] Srivastava, B.: An Affective Heuristic for Minimising Makespan on Unrelated Parallel Machines. Journal of the Op. Research Soc., Vol. 49, 1998, No. 8, pp. 886–894.

[12] Taillard, E.: Robust Tabu Search for the Quadratic Assignment Problem. Parallel Computing, Vol. 17, 1991, pp. 443–455.

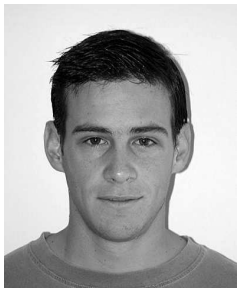
[13] Thesen, A.: Design and Evaluation of Tabu Search Algorithms for Multiprocessor Scheduling. Journal Heuristics, Vol. 4, 1998, No. 2, pp. 141–160.

[14] Talbi, E.-G.—Zomaya, A.: Grids for Bioinformatics and Computational Biology. John Wiley & Sons, USA, 2007.

[15] Xhafa, F.—Alba, E.—Dorronsoro, B.—Duran, B.: Efficient Batch Job Scheduling in Grids Using Cellular Memetic Algorithms. Journal of Mathematical Modelling and Algorithms, Vol. 7, 2008, No. 2, pp. 217–236, Kluwer.

Fatos Xhafa received his Ph. D. in computer science from the Technical University of Catalonia, UPC (Barcelona, Spain) in 1998. He is currently Associate Professor and member of the ALBCOM Research Group of LSI department. His current research interests include parallel algorithms, combinatorial optimization, approximation and meta-heuristics, distributed programming, Grid and P2P computing. He has published in leading international journals and conferences and has served in the Organizing Committees of many conferences and workshops. He served as Organizing Chair of ARES 2008, PC chair of CISIS 2008, Workshops co-chair of CISIS 2008 and General co-chair of HIS 2008 conferences. Presently he is Workshop Chair of CISIS-2009.

Javier Carretero received the M. Sc. degree in computer engineering from UPC in 2005. Since April 2006, he is a research scientist of the Intel Barcelona Research Center. He is currently doing the PhD at UPC on the area of resiliency. His main research interests include processor micro-architecture, hardware reliability, and lightweight on-line testing, Grid computing, optimization, failure detection and networking systems.



Bernabé Dorronsoro received the degree in engineering (2002) and the Ph. D. in computer science (2007) from the University of Málaga (Spain), and he is currently working as scientific collaborator at the University of Luxembourg. His main research interests include grid computing, ad hoc networks, the design of new efficient meta-heuristics, and their application for solving complex real-world problems. He has been member of the organizing committees of several conferences and workshops, and he usually serves as reviewer for leading impact journals and conferences.

Enrique Alba received his degree in engineering and Ph. D. in computer science in 1992 and 1999, respectively, from the University of Málaga (Spain). He works as a Professor in this university. He leads a team of 7 doctors and 8 engineers in the field of complex optimization. In addition to the organization of international events he has offered dozens doctorate courses, multiple seminars in more than 20 international institutions and has directed several research projects (4 with national funds, 3 in Europe and numerous bilateral actions). He also works as invited professor at INRIA and the University of Luxembourg. He is editor in 13 international journals and one book series of Springer-Verlag. He has published in journals indexed by Thomson ISI, papers in LNCS, and in refereed conferences.