Design and Evaluation of Tabu Search Method for Job Scheduling in Distributed Environments

Fatos Xhafa and Javier Carretero
Dept. of Languages and Informatics Systems
Polytechnic University of Catalonia, Spain
fatos@lsi.upc.edu

Enrique Alba
Dept. of Languages and Computer Science
University of Málaga, Spain
eat@lcc.uma.es

Bernabé Dorronsoro
Faculty of Science, Technology and Communication
University of Luxembourg, Luxembourg
bernabe.dorronsoro@uni.lu

Abstract

The efficient allocation of jobs to grid resources is indispensable for high performance grid-based applications. The scheduling problem is computationally hard even when there are no dependencies among jobs. Thus, we present in this paper a new tabu search (TS) algorithm for the problem of batch job scheduling on computational grids. We consider the job scheduling as a bi-objective optimization problem consisting of the minimization of the makespan and flowtime. The bi-objectivity is tackled through a hierarchic approach in which makespan is considered a primary objective and flowtime a secondary one. An extensive experimental study has been first conducted in order to fine-tune the parameters of our TS algorithm. Then, our tuned TS is compared versus two well known TS algorithms in the literature (one of them is hybridized with an ant colony optimization algorithm) for the problem. The computational results show that our TS implementation clearly outperforms the compared algorithms. Finally, we evaluated the performance of our TS algorithm on a new set of instances that better fits with the concept of computational grid. These instances are composed of a higher number of –heterogeneous– machines (up to 256) and emulate the dynamic behavior of these systems.

1. Introduction

Computational Grid (CG) is a new distributed computing paradigm for the development of large-scale distributed applications [11, 12, 13]. One of the main objectives of CGs is to provide computational frameworks that support applications, which could benefit from the large computing potential of such distributed infrastructures. In fact, almost immediately after the introduction of CGs, the grid approach was validated in practice by several projects such as NetSolve [8] and MetaNeos including applications for stochastic programming [17] and optimization [15, 24].

CGs currently represent a very successful approach for large-scale distributed real-world applications; numerous examples of such applications are being reported in the literature [18, 22]. Nonetheless, the grid computing paradigm is raising important issues regarding the development of large-scale distributed applications. One such issue is the efficient dynamic allocation of jobs to geographically distributed resources. Although the family of scheduling problems is one of the most studied ones by the optimization research community, application of the available approaches to the job scheduling on CGs is not straightforward, as it differs significantly from conventional scheduling in distributed systems. This can be explained by the fact that scheduling in grid systems adds new features not present in conventional scheduling problems. The following features could be distinguished:

- *Heterogeneity of jobs*: jobs submitted to the grid system could be originated by different users and applications. In general, the grid system is not aware of the type of jobs being submitted. Moreover, jobs may have different workloads, could require different resource capacity, have different amounts of associated data, etc.
- *Job restrictions*: apart from the usual scheduling characteristics such as release date, jobs can have other restrictions on the type of resources needed to solve them and incompatibilities could exist.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

- *Multi-objectivity*: several objectives, which could be contradictory, are to be optimized. Examples of these objectives include makespan, flowtime, matching proximity, or resource utilization. Other objectives could be considered if the resources were to be used in *pay-per-use* mode, requiring, for instance, the maximization of user's benefit.
- *Large-scale*: grids are expected to be large or very large in size and a large number of jobs could be submitted by independent users and applications.

It should be noted that, unlike traditional schedulers, any grid scheduler will be running as long as the grid itself exists, since jobs can be submitted at any time and the information on the current state of resources and jobs should be kept up to date. One important implication of this is that any grid scheduler must achieve allocations[1] of jobs to resources in very short times and must be robust in order to adapt itself to the changes of in the grid.

One simple yet very important version of the job scheduling is that of scheduling independent jobs. Given the large size as well as the decentralized nature of the grids, this type of scheduling arises naturally when a number of independent users/applications submit their jobs to the grid; also, it appears in a family of applications known as *parameter sweep* applications [9, 10], which consist of many independent jobs that could use files for input and output (e.g. Monte-Carlo simulations).

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 the job scheduling on computational grids for the bi-objective case[2], namely, 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 *ingredients* (sub-algorithms). The TS implementation presented here explores this flexibility and, thus by carefully designing and implementing the TS sub-algorithms and tuning of TS parameters, our TS implementation is able to outperform other known heuristic approaches. The experimental study was carried out, on the one hand, by using a static benchmark [6], and, on the other hand, by using a prototype of a grid simulator. We study the static benchmark for comparison purposes, and then, after demonstrating the validity of our method, we tackle a much more realistic benchmark we defined folowing the directives in [7]. Our implementation achieves very fast makespan reductions, thus making the resulting scheduler adequate for grid applications. It is noted that some existing TS-based approaches in the literature require long computing times to achieve significant makespan reductions, which is usually prohibitive for a grid system. Moreover, the reported results for the problem in the literature are limited to rather "small" size instances, consisting of a dozen of machines and hundreds of jobs. Therefore, medium, large and very large size instances are generated and TS implementation is tested, as this is more realistic for grid systems. Finally, we also address the issue of studying the performance of the TS scheduler in a dynamic environment.

The paper is organized as follows. We give in Section 2 the description of job scheduling in CGs considered in this work. TS and its particularization for the problem studied are given in Section 3. The experimental study is presented in Section 4. We summarize in Section 5 the most important results of this work and indicate directions for future work.

2 Problem definition

We present in this section the problem of job scheduling in computational grids. For making realistic simulations, it is necessary that the problem captures the most important features of CGs. We are using in this paper the model of Braun et al. [6], that simulates heterogeneous distributed environments and allows to introduce characteristics of both jobs and resources of the grid system. Further, as we will see shortly, several optimization criteria can be defined in Braun's model, capturing thus the multi-objectivity feature of the problem. In this model a collection of jobs with no inter-dependencies is considered for allocation of resources. The problem 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 using resource m. One possible way to compute the entries $ETC[j][m]$ 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. Examples of computation of the job workloads can be found for the Cornell Theory Center [16] or the Parallel Workload Archive [1].

Using the ETC matrix model, an instance of the job scheduling, at a given instant of time, can be defined as follows:

- A number of independent *jobs* to be allocated to grid resources. Each job has to be processed entirely in a single resource and is not preempted (once started, a job runs until completion).

[1]We do not consider the co-allocation feature, that is, the simultaneous use of several resources to solve a single job is not considered.

[2]Resource utilization has also been considered, though it is not reported here.

- A number of *machines* that are candidate to participate in 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 will have finished the previously assigned jobs. This parameter measures the previous workload of a machine.
- The ETC matrix of 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.

Solving this problem is in fact a multiobjective task, since the fitness of a schedule can be measured using several optimization criteria, such as minimizing the *makespan* (that is, the finishing time of the latest job), the *flowtime* (i.e., 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. We consider that the most important criterion is that of minimizing the *makespan*, and certainly, this is the most reported parameter in scheduling studies of distributed systems. Additionally, we are also considering in this work the minimization of the *flowtime* of the grid system as a secondary criterion. These two criteria are formally defined as follows:

- *makespan*: $\min_{S_i \in Sched}\{\max_{j \in Jobs} F_j\}$ and,
- *flowtime*: $\min_{S_i \in Sched}\{\sum_{j \in Jobs} F_j\}$,

where F_j denotes the time when job j finalizes and $Sched$ is the set of all possible schedules. Note that the makespan is not affected by any particular job execution order in a concrete resource, while in order to minimize the flowtime of a resource, the assigned 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 to in detriment of the other, especially for near-optimal schedules. This is especially evidenced for schedules which are close to optimal.

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. In this second step, the value for makespan can not be worsened. All the details on the proposed algorithms can be found in Section 3.

3 Tabu Search for Scheduling on Computational Grids

In this section, we present the TS algorithm proposed for solving the job scheduling problem described in Section 2. The TS method was introduced by Glover [14] as a high-level algorithm that uses other specific heuristics to guide the search; the objective is to perform an intelligent exploration of the search space that would eventually allow to avoid getting trapped into local optima. The template we have used for designing our TS algorithm is provided in Figure 1. This code has shown to be very effective for several problems [2, 3, 4, 5].

```
begin
  Compute an initial solution $s$; let $\hat{s} \leftarrow s$;
  Reset the tabu and aspiration conditions.
  while not termination-condition do
    Generate a subset $N^*(s) \subseteq N(s)$ of solutions such
      that (none of the tabu conditions is violated)
      or (the aspiration criteria hold);
    $s \leftarrow$ best $s' \in N^*(s)$ in terms of cost function;
    if improvement($s'$,$\hat{s}$)) then $\hat{s} \leftarrow s'$; endif;
    Update the recency and frequency;
    if (intensification condition) then
      Perform intensification procedure;
    endif;
    if (diversification condition) then
      Perform diversification procedures;
    endif;
  endwhile;
  return $\hat{s}$;
end;
```

As it can be seen from the template of Fig. 1, one of the distinguishing features of TS versus other heuristics is the use of an *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 whole complete exploration process. Additionally, when designing a TS algorithm for a specific problem we need to specify some inner heuristics like the *local search*, used for exploring the neighborhood of a solution, the *tabu status and aspiration criteria*, for managing the list of recently visited solutions, and the *intensification and diversification procedures*, for appropriately managing the exploration/exploitation tradeoff on the search space. In the following sections we describe the TS algorithm used in this work.

We consider a schedule as a vector of job-machine allocations of size nb_jobs, in which $schedule[i]$ indicates the machine where job i is assigned to. Thus, the values of this vector are natural numbers in $[1, nb\_machines]$. Note that in this representation a machine number can appear more than once.

Two types of movements [23] are considered for this representation: *transfer* and *swap*. Transfer moves a job from one machine to another one and swap interchanges two jobs assigned to different machines. Note that these types of movements take into account the specific need of load balancing. Swap leaves the number of jobs assigned to the machines constant, while transfer changes it. Note also that a swap can be achieved by two consecutive transfers.

We have used *Min-Min* method to generate the starting solution. Min-Min starts by computing a matrix of values $completion[i][j]$ for any job i and machine j based on $ETC[i][j]$ and $ready[j]$ values ($completion[i][j] = ETC[i][j]+ready[j]$). For any job i, the machine m_i yielding the earliest completion time is computed by traversing the ith row of the completion matrix. Then, job i_k with the earliest completion time is chosen and assigned to the previously computed machine m_k. Next, job i_k is removed from the set of jobs to do ($Jobs$) and the values $completion[i][j]$ for each i in $Jobs$ and machine m_k are updated. The process is repeated until no job remain to be assigned.

Both short and long term memories have been used in our TS algorithm. For the *recency* memory, a matrix TL ($nb\_jobs \times nb\_machines$) is used to maintain the tabu list[3] in which $TL[j][m]$ indicates the number of the last iteration in which job j was assigned to machine m. Whenever the value of $TL[j][m]$ changes, this means that a new assignment is made due to a movement (e.g., transfer or swap). In this case, the original assignments before the application of the movement are therefore made tabu. A transfer movement effects just one entry of the TL matrix while a swap movement effects two entries. In addition to this TL, a tabu hash table (TH) is maintained in order to know which solutions (*hash Ids*) have been already visited. Note that TH is used as a complementary information to TL in the sense that TL is able to detect only movements which have been assigned tabu status recently, while TH can give such information for longer time periods. Thus, using this information together with information on tabu movements we are able to filter even more movements that lead to previously visited solutions.

Regarding the *frequency* memory, a matrix $frequency$ ($nb\_jobs \times nb\_machines$) is used, in which $frequency[j][m]$ indicates how many times job j has been assigned to machine m. This way, the frequency table we keep indicates for any task how many times it has been assigned to different machines. This memory is used in the intensification phase (see Section 3.6). Finally, as is standard practice, a user-specified number of elite solutions is kept.

[3]This is adopted from Taillard [21].

Aspiration criteria are used to remove the tabu status of movements. Initially we used two well-known criteria to determine the aspiration level of movements:

- Fitness based criterion: it consists of accepting a tabu movement if it yields to a better solution. Thus, for a given solution s, the set $A_{better}(s)$ of its neighbor solutions s' that are better than s according to the evaluation criterion are accepted for evaluation: $A_{better}(s) = \{s' \mid \texttt{improvement}(s, s')\}$.
- Using the TL matrix[4]: the set of aspiring solutions of s at iteration k is computed as $A(s) = A_{transf}(s) \cup A_{swap}(s)$:

$$\begin{aligned}
A_{transf}(s) = \{s' \mid s' = s \oplus m_{transf}(t_i, m_j), TL[i][j] + \\
asp\_value) \leq k\} \\
A_{swap}(s) = \{s' \mid s' = s \oplus m_{swap}(t_i, t_j), \max(TL[i][s(j)], \\
TL[j][s(i)]) + asp\_value) \leq k\},
\end{aligned}$$

where asp_value is a user-specified value indicating the minimum number of iterations after which a tabu movement can aspire. Thus, for a given solution s and a tabu movement m, if the solution s' obtained by applying m to s belongs to $A_{better} \cup A_{transfer}(s) \cup A_{swap}(s)$ then m is accepted for evaluation.

Notice that the above aspiration criteria are somehow restrictive (the first criterion uses a global fitness value and the second one takes into account the lifetime of a tabu movement). Therefore, we have defined another aspiration criterion based on the value of the local makespan, that is, the makespan relative to a movement (remember that the makespan is the primary objective in our hierarchical approach). This criterion is defined as follows:

$$\begin{aligned}
A'_{transf}(s) = \{s' \mid s' = s \oplus m_{transf}(t_i, m_j), \\
makespan(s') < best\_makespan(s[i], [j])\} \\
A'_{swap}(s) = \{s' \mid s' = s \oplus m_{swap}(t_i, t_j), \\
makespan(s') < best\_makespan(s[i], s[j])\},
\end{aligned}$$

where $best\_makespan(\cdot, \cdot)$ is a $nb\_jobs \times nb\_machines$ matrix whose i, j position indicates the smallest makespan value achieved by moving jobs from machines m_i and m_j. This criterion actually turned out to give better results and, actually, larger sets of aspiring solutions were obtained.

[4]Similar to Taillard's criterion in his TS for QAP problem.

TS is a local search heuristic, therefore the quality of encountered solutions largely depends on the effectiveness and efficiency of the neighborhood exploration. The neighborhood of a solution is determined by two types of movements, namely transfer and swap, which are applied with equal probability. Scheduling in grids is a large-scale problem, so one important issue here is to reduce the size of the neighborhood since full exploration could penalize the overall time of the TS. Thus, in order to make a reasonable trade-off between the neighborhood size and the quality of neighboring solutions, we use the load balancing as a criterion: transfers and swaps are made among jobs assigned to most-loaded machines and jobs assigned to less-loaded machines, as defined next.

In fact, even after reducing the number of candidate machines for transfers and swaps, the remaining neighborhood can still be large, so we also put upper bounds on the maximum number of transfers and swaps that can be evaluated. Thus, we defined four parameters (of user-specified values): (a) $max\_load\_factor \in [0,1]$ to compute the set of most overloaded machines with respect to the makespan of the current schedule (the overloaded machines are those for which $completion[m] > max\_load\_factor \cdot local\_makespan$); (b) min_load_factor used to compute the set of less overloaded machines (machines are sorted according to their completion time and the first $min\_load\_factor \cdot nb\_machines$ machines are considered); (c) $max\_transfs$ for the maximum number of transfer movements; and (d) max_swaps for the maximum number of swap movements.

For a given solution s, candidate neighbor solutions s' are evaluated according to a steepest-descent criterion (the best movement w.r.t. improvement criterion) or mildest-ascent (the least worst movement) in case no better solutions are encountered among neighbor solutions. The improvement criterion can be defined in different ways, such as minimizing the makespan, minimizing the difference between the completion times of s and s', or minimizing the completion time of the most overloaded machine. Here, we have chosen the minimization of completion time of the most overloaded machine when restricted to the new machines involved in the neighbor solution. Thus, again, we preferred a local optimality criterion rather than a global one. Formally, being M_s the machines involved in s, the criterion is $\min_{s'}\{\max_{m \in M'} completion[m]\}$ where $M' = \{s^{-1}(m) \neq s'^{-1}(m) \mid m \in M_s\}$, that is, the set of machines involved in s effected by the movement that leads to s'.

The intensification procedure is activated when there is evidence that the region of the current solution could contain good solutions. Thus, the main course of the search is temporarily changed in order to carry out a thorough exploration of the region that contains the current solution. Usually, this more deeper exploration is done by means of rewarding (attributes of) the current solution, thus forcing their presence in new solutions. Three different strategies for intensifying the search are considered and a combination of them is finally applied.

- *Using elite solutions*. The frequency table obtained from elite solutions is used for rewarding the most promising attributes of solutions. For each job, its most frequent assignment is chosen with probability .75, while in other case (probability .25) it is assigned using a roulette-wheel.
- *Changing temporarily* the values for the maximum and minimum load factor parameters. Essentially, instead of using the user-specified values for these parameters, we compute them 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 (see Section 3.5). Then, the neighborhood is defined by these new values.
- *Changing the structure of the neighborhood*. Instead of applying transfers and swaps with equal probability, we apply first a sequence of swaps followed by just one transfer. The method is as follows: two machines m_1 and m_2 are randomly chosen, and next, we first apply all possible swaps of jobs in m_1 with those in m_2 and then we apply just one transfer from m_2 to m_1. A strictly steepest–descent criterion (w.r.t. completion times) is used for the evaluation.

As we previously introduced, these three strategies are combined following a hierarchical approach: we start by applying the first strategy until no improvements are possible, and then we proceed by applying the second and the third ones.

Diversification is conceptualized in two forms (see also [2, 5]): *soft* diversification, which promotes the search in a new region that is "not far" from the current one, and *strong* diversification, which is a re-start search. The idea is to avoid using only the abrupt interruption, which is a typical application use of this procedure in the TS method.

To this end, we have implemented three different forms of soft diversification in addition to the strong diversification, as explained below.

Using the job distribution. This idea is taken from Hübscher and Glover [?], and it is also known as *influential diversification*. Essentially, we try to redistribute the jobs to machines in such a way that any machine is assigned long and short jobs. In other words, it is considered potentially problematic that some machines have an "excessive" number of long[5] jobs while it is assigned a large number of many short jobs to others. Thus, by redistributing the jobs among these machines would imply a slight perturbation of the schedule with the aim of obtaining a better one. To this end, a kind of job distribution factor is computed using the expected time to compute values in order to identify two machines: one with the largest number of short jobs and another one with the longest jobs. Then, a subset of jobs from the first is exchanged with a set of jobs from the second using the minimum completion time strategy.

Using penalization of ETC values. Penalizing attributes of a solution is one of the most commonly used forms of diversification. More precisely, we use the long term memory (*frequency*) to penalize the corresponding $ETC[j][m]$ values of most frequent job-machine assignments.

Freezing jobs. This is also another penalization method. In this case, the jobs that have most frequently changed their assignments are frozen. This information is again obtained from the long term memory. Freezing a job at an iteration is simply done by assigning tabu status to all movements that involve the job for the concrete iteration; after the diversification is carried out, the tabu status of these job(s) is cancelled.

Strong diversification. In this work, we have discarded the possibility of re-starting the search from a new initial solution in order to avoid introducing too much diversity; instead, 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 of the experimental study for the proposed TS implementation. In order to measure the quality of our algorithm, we compare it versus some other algorithms in the literature on a classical benchmark on job scheduling on grid computing (Section 4.1). After validating our algorithm, we use it for solving more realistic instances we generated for this work (Section 4.2).

[5]Long and short refer to large and small ETC values.

The objective of this section is to show the quality of the TS algorithm we propose in this work. For that, our algorithm is compared versus both a TS and an ant colony optimization algorithm hybridized with a TS (ACO+TS). These two compared algorithms were proposed by Ritchie and Levine [19]. Thus, we selected the same benchmark used in that work and compare all the algorithms in terms of makespan (no results for flowtime were presented in [19]). The instances of this benchmark are classified according to three parameters (job heterogeneity, machine heterogeneity, and consistency) into 12 different types of ETC matrices, each of them these consisting of 100 instances. For all instances, the number of jobs is 512 and the number of machines is 16. Instances are labelled as $u\_x\_yyzz.k$ where u means uniform distribution (used in generating the matrix), 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), respectively, and k is used to number instances of the same type. In this benchmark, 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 of the jobs faster than m_j. Inconsistency means that a machine is faster for some jobs and slower for some others, while an ETC matrix is considered semi-consistent if it contains a consistent sub-matrix.

The parametrization used in the proposed TS is shown in Table 1. As it can be seen, we used the Min-Min method for generating the initial solution. The size of the tabu hash table (TH) is set to 918133, which is a high number and non divisor of 20, as it is recommended by Srivastava [20]. The maximum number of iterations a solution remains tabu (`max_tabu_status`) is chosen uniformly from the interval $[\texttt{nb\_machines}, 2\cdot\texttt{nb\_machines}]$, and the maximum number of successive iterations without improvements of the current solution implying the activation of the intensification (`max_repetitions`) is fixed to $4\ln(\texttt{nb\_jobs})\cdot\ln(\texttt{nb\_machines})$. The number of iterations of a diversification (`nb_diversifications`) and an intensification (`nb_intensifications`) are set to $\log_2(\texttt{nb\_jobs})$. The `max_load_factor` and `min_load_factor` parameters are used to identify most and less overloaded machines, respectively, and their values are set to 1.0, and there is no limit for the maximum number of movements (`max_nb_swaps` , `max_nb_transfs`) allowed while exploring the neighborhood. Finally, we consider a number of 30 elite solution, and a value of $(\texttt{max\_tabu\_status}/2)-\log_2(\texttt{max\_tabu\_status})$ for the minimum number of iterations after which a tabu movement can aspire. The algorithm runs for 100 seconds.

We give in Table 2 the computation results for makespan value of our TS implementation and the TS and TS+ACO

`start_choice`	Min-Min method
`tabu_size`	918133
`max_tabu_status`	32
`max_repetitions`	69
`nb_diversifications`	8
`nb_intensifications`	8
`max_load_factor`	1.0
`min_load_factor`	1.0
`max_nb_swaps`	∞
`max_nb_transfs`	∞
`nb_iterations`	8192
`elite_size`	30
`aspiration_value`	20
`max_time_to_spend`	100 seconds

algorithms by Ritchie and Levine in [19]. In the case of our TS algorithm, each reported value is the best makespan value out of 10 executions. In order to demonstrate the robustness of our proposal, we include in the last two columns of Table 2 the average makespan obtained and its deviation w.r.t. our best makespan value.

As can be seen in Table 2, our TS is more effective than the Ritchie's TS and ACO+TS algorithms. More precisely, our implementation outperforms Ritchie's ones for 9 out of 12 considered instances. For the remaining three instances the improvement over our TS is very small. We believe that this better performance of our TS implementation is due to a better embedding of problem specific knowledge into the problem-dependent procedures of the TS method. Moreover, our reduced execution time of 100 seconds (fixed by the stopping criterion of the algorithm) is far lower than Ritchie's execution times, which is probably due to the use of more efficient data structures in our implementation. At this point, it should be noted that the execution time of the scheduler in a dynamic environment, such as grid systems, is a critical factor. Thus, the short execution times achieved by our TS implementation show that the TS method yields to fast and significant reductions of makespan and is thus very suitable for grid schedulers.

As an extension to the experiments carried out in the previous section, we proceed here to apply our TS to a more realistic benchmark (ranging from 32 to 256 machines) in order to evaluate the performance in more realistic scenarios. A description of the benchmark used for this brief study can be found in [7], and its main feature (in addition to the larger instances size) is that several parameters are dynamically changing, such as the number and the kind of available resources and the jobs to be scheduled.

We give in Table 3 makespan and flowtime values for instances generated using this dynamic benchmark. We observe that makespan value increases slowly as the instance size is doubled while the flowtime increases considerably (almost doubled). Additionally, we compare in Table 3 the performance of our TS versus a steady-state genetic algorithm (ssGA) proposed by Carretero and Xhafa in [7]. The algorithms are compared only in terms of the makespan because no values for flowtime were reported in the referred work. As it can be seen, our new TS implementation outperforms the compared algorithm for the studied instances, as it already 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 $\pm$ %C.I (0.95)		Flowtime $\pm$ %C.I (0.95)
	TS	ssGA	TS
32	3969016.8$_{\pm 0.4\%}$	4063425.5$_{\pm 0.8\%}$	1047867441.0$_{\pm 0.9\%}$
64	3970894.0$_{\pm 0.4\%}$	3994804.9$_{\pm 0.9\%}$	2102952840.2$_{\pm 0.8\%}$
128	3980381.4$_{\pm 0.4\%}$	3995162.0$_{\pm 1.3\%}$	4199261733.0$_{\pm 0.8\%}$
256	3972429.0$_{\pm 0.4\%}$	4009852.0$_{\pm 1.8\%}$	833351728.9$_{\pm 0.9\%}$

5 Conclusions and future 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 application for solving large-scale optimization problems 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, the TS has also been tested in a more realistic (dynamic) framework, outperforming also previous approaches.

In our further work we would like to complete the experimental study of the TS scheduler in the dynamic setting. Also, we would like to address parallelization of the TS implementation using existing parallel models for the method in the literature.

Acknowledgments

Research partially supported by ASCE TIN2005-09198-C02-02, FP6-2004-IST-FETPI (AEOLUS) and MEC TIN2005-25859-E Projects. E. Alba acknowledges partial

Instance	Ritchie's TS	Ritchie's ACO+TS	Our TS	Our TS (average)	Our TS (dev.)
u_c_hihi.0	7568871.83	7497200.85	**7448640.471**	7458864.453	0.124%
u_c_hilo.0	154644.48	154234.63	**153263.333**	153438.078	0.063%
u_c_lohi.0	245981.55	244097.28	**241672.657**	242385.384	0.309%
u_c_lolo.0	5202.51	5178.44	**5154.980**	5155.783	0.014%
u_i_hihi.0	3021155.10	2947754.12	2957854.074	2959029.352	0.041%
u_i_hilo.0	74400.68	73776.24	**73692.853**	73734.844	0.047%
u_i_lohi.0	104309.12	102445.82	103865.666	103867.134	0.011%
u_i_lolo.0	2580.62	2553.54	**2552.070**	2559.961	0.126%
u_s_hihi.0	4248200.21	**4162547.92**	4168795.890	4181985.827	0.260%
u_s_hilo.0	97711.72	96762.00	**96180.850**	96432.137	0.203%
u_s_lohi.0	126115.39	123922.03	**123407.442**	123600.512	0.087%
u_s_lolo.0	3505.69	3455.22	**3450.532**	3454.022	0.064%

support from the Spanish MEC and FEDER under contract TIN2005-08818-C04-01 (the OPLINK project).

References

[1] The hebrew university parallel systems lab., parallel workload archive, http://www.cs.huji.ac.il/labs/parallel/workload/.

[2] E. Alba, F. Almeida, M. Blesa, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, G. Luque, J. Petit, C. Rodríguez, A. Rojas, and F. Xhafa. Efficient parallel LAN/WAN algorithms for optimization. the Mallba project. *Parallel Computing*, 32(5-6):415–440, 2006.

[3] M. Blesa, L. Hernandez, and F. Xhafa. Tabu search for 0-1 multidimensional knapsack revisited: choosing internal heuristics and fine tuning of parameters. In *12th Young Operational Research Conference*, 2001.

[4] M. Blesa, L. Hernandez, and F. Xhafa. Parallel skeletons for tabu search method based on search strategies and neighborhood partition. In *4th International Conf. on Parallel Processing and Applied Mathematics (PPAM'01)*, volume 2328 of *LNCS*, pages 185–193. Springer, 2002.

[5] M. Blesa, J. Petit, and F. Xhafa. Generic parallel implementations for tabu search. To appear, 2006.

[6] H. Braun, T. D. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. 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.

[7] J. Carretero and F. Xhafa. 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.

[8] H. Casanova and J. Dongarra. Netsolve: Network enabled solvers. *IEEE Computational Science and Engineering*, 5(3):57–67, 1998.

[9] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop*, pages 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 *Proc. of the 2000 ACM/IEEE Conf. on Supercomputing (CDROM)*, pages 75–76. IEEE Press, 2000.

[11] I. Foster. *What is the grid? A three point checklist*. White Paper, 2002.

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

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

[14] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5:533–549, 1986.

[15] J. Goux and S. Leyffer. Solving large MINLPs on computational grids. *Optimization and Engineering*, 3:327–346, 2002.

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

[17] L. Linderoth and S. Wright. Decomposition algorithms for stochastic programming on a computational grid. *Computational Optimization and Applications*, 24:207–250, 2003.

[18] F. Luna, A. Nebro, and E. Alba. Observations in using grid-enabled technologies for solving multi-objective optimization problems. *Parallel Computing*, 32:377–393, 2006.

[19] 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.

[20] B. Srivastava. An affective heuristic for minimising makespan on unrelated parallel machines. *Journal of the Op. Research Soc.*, 49(8):886–894, 1998.

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

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

[23] A. Thesen. Design and evaluation of tabu search algorithms for multiprocessor scheduling. *Journal Heuristics*, 4(2):141–160, 1998.

[24] S. Wright. Solving optimization problems on computational grids. *Optima*, 65, 2001.