Improving QoS in Computational Grids through Schedule-based Approach

Dalibor Klusáček and **Hana Rudová**
Faculty of Informatics, Masaryk University
Botanická 68a, Brno 602 00
Czech Republic
{xklusac, hanka}@fi.muni.cz

Abstract

While Grid users are often interested in satisfaction of their *Quality of Service (QoS)* requirements, these cannot be satisfactory handled by commonly used *queue-based* approaches. This paper concentrates on the application of *schedule-based* methods which allows both efficient handling of QoS requirements as well as traditional machine usage objective. An incremental application of our methods reacting on dynamic character of the problem allows to achieve reasonable runtime. Even more, we show that schedule-based methods significantly outperforms queue-based approaches by means of weighted machine usage reflecting heterogeneity of the resources common for Grid environments.

A new formalized description of two *schedule-based* methods is introduced to schedule dynamically arriving jobs onto the machines of the computational Grid. Earliest Gap—Earliest Deadline First (EG-EDF) policy fills earliest gaps (EG) in the known schedule with newly arriving jobs, incrementally building a new schedule. If no gap for a coming job is available EDF policy places the new job into the existing schedule. Tabu search algorithm is used to further optimize the schedule. It moves selected jobs into earliest suitable gaps again. Proposed methods are compared with some of the most common queue-based scheduling algorithms like FCFS (First Come First Served), EASY backfilling, and Flexible backfilling.

Introduction

The purpose of *Grid technology* is to manage large and heterogeneous computer environment that will allow an easy access to Grid resources for various users, by means of allowing them to submit their jobs into the system, guaranteeing them *nontrivial QoS* while hiding the complexity of the system itself by providing powerful but simple interfaces for the end user of the Grid (Foster and Kesselman 1998). Moreover, not only users but also resource owners should be satisfied—in this case keeping the resource usage reasonably high is usually very important. Therefore, *multi-objective criteria* have to be met. To meet these goals sophisticated and automated scheduling techniques should be applied. On the other hand, Grid is highly dynamic, distributed, and heterogeneous environment so the scheduling is an extremely difficult task if good performance, QoS or robustness is required.

Current scheduling techniques applied in Grids are mostly based on the queueing systems of various types which are designed with respect to specific needs of Grid technology. Present systems like PBS (Jones 2005), LSF (Xu 2001), Sun Grid Engine (Gentzsch 2001), Condor (Thain, Tannenbaum, and Livny 2005) together with complex Grid management systems such as GridWay (Huedo, Montero, and Llorente 2005) or Moab (Clu 2008) represent de facto standard solutions. Still, they are all using simple queue-based scheduling policies.

While single objectives can often be well satisfied with a proper queue-based policy, complex objectives including e.g., response time, slowdown, deadlines, resource utilisation, etc., are hard to achieve by a queue-based solution, especially for common user. Nowadays, users are often forced to cheat when looking for good performance of their applications, e.g., by bypassing the scheduling system through direct logging onto specific machine and starting their jobs from the command line. In current systems the concept of *advanced reservation* of resources is often implemented to guarantee certain QoS level, however this functionality is often restricted by Grid administrators since the queue-based schedulers are unable to manage large number of reservations efficiently. In fact, when the amount of jobs requesting reservation exceeds certain level the system usage drops very quickly and a starvation of other jobs without reservation often appears (Smith, Foster, and Taylor 2000).

In dynamic environments, such as Grids, resource may change, jobs are not known in advance and they appear while others are running. *Schedule-based approach* allows precise mapping of jobs onto machines in time. This enables us to use advanced scheduling algorithms (Pinedo 2005) such as local search-based methods (Glover and Kochenberger 2003) to optimize the schedule. Due to their computational cost, these approaches were mostly applied to static problems, assuming that all the jobs and resources are known in advance which allows to create schedule for all jobs at once (Armentano and Yamashita 2000; Baraglia, Ferrini, and Ritrovato 2005). CCS (Hovestadt et al. 2003) as well as GORBA (Süß et al. 2005) are both advanced Grid re-

source management systems that use schedule instead of a queue(s) to schedule workflows (GORBA), or sequential and parallel jobs while supporting the advanced reservations (CCS). GORBA uses simple policies for schedule creation and an evolutionary algorithm for its optimisation while CCS uses FCFS, Shortest/Longest Job First policies when assigning jobs into the schedule and a backfill-like policy that fills gaps in the constructed schedule. Both CCS and GORBA re-compute the schedule from scratch when a dynamic change such as job arrival or machine failure appears. It helps to keep the schedule up to date, however for large number of jobs this approach may be quite time consuming as was discussed in case of GORBA (Stucky et al. 2006). Several papers (Abraham, Buyya, and Nath 2000; Subrata, Zomaya, and Landfeldt 2007) propose local search based methods to solve Grid scheduling problems. The schedule is kept valid in time without total re-computation, however no experimental evaluation was presented in (Abraham, Buyya, and Nath 2000), and (Subrata, Zomaya, and Landfeldt 2007) does include resource changes but no dynamic job arrivals.

In this paper we provide a formalized description of our schedule-based solution. Brief description of some of the ideas used was already given in our previous work (Klusáček et al. 2008). Our approach allows to efficiently schedule *dynamically arriving* jobs onto the machines of a computational Grid. The initial schedule is generated by a simple and fast EG-EDF policy and then periodically optimized with a Tabu search algorithm. In comparison with other approaches (Hovestadt et al. 2003; Süß et al. 2005), we use the *policy* as well as *local search* in an *incremental* fashion (Klusáček et al. 2008). It means that last computed schedule is used as the starting point for building a new and up to date schedule. This leads to a reasonable computational cost since the schedule is not rebuilt from scratch. Moreover, we propose a multi-criteria objectives which focus on providing nontrivial QoS to the Grid users, while satisfying the system administrator's requirements as well. User QoS requirements are expressed by the objective function focusing on maximizing the number of jobs that meet their deadline (Capannini et al. 2007), while system administrators needs are expressed by a machine usage criterion. The success of the solution is based on an efficient method which detects and fills existing gaps in the schedule with suitable jobs. It allows us to increase both the QoS and machine usage by limiting fragmentation of the processor time.

Our solution was evaluated through set of experiments performed in Alea simulator (Klusáček, Matyska, and Rudová 2008) against typical queue-based scheduling algorithms like FCFS, EASY backfilling and Flexible backfilling (Techiouba et al. 2008).

Problem Description

In our study we expect fixed set of m machines but we allow changes in the set of jobs. As the time is running, new jobs may appear and processing of other jobs is meanwhile completed. Newly arriving jobs are placed into the schedule which define where and when they will be executed. Each job is characterized with the release date (arrival time) r_j representing the time when the job appears in the system. Job deadline d_j is understood as a desired job completion time which should be kept. Job j has known processing time $p_{i,j}$ which depends on the CPU speed of machine i. Job also requires $R_{i,j}$ number of CPUs for its execution ($R_{i,j} > 0$). Resources are computational machines with known capacity R_i representing the number of CPUs. All CPUs within one machine has the same speed s_i representing the number of operations per second. Different machines may have different speeds and number of CPUs. Machines use the Space Sharing processor allocation policy which allows parallel execution of k jobs on machine i if $R_i \geq \sum_{j=1}^{k} R_{i,j}$.

Various objective functions can be considered such as makespan (C_{max}) or average flow time. Our scheduler aims to maximize both the *machine usage* and the number of jobs with *respected deadlines* (Capannini et al. 2007). A higher machine usage fulfills resource owner's expectations, while a higher number of non delayed jobs guarantees a higher QoS provided to the users. This value is represented by *unit penalty* function $U = \sum_{j=1}^{n} U_j$ where $U_j = 1$ if $C_j > d_j$ holds otherwise U_j equals to 0. Since C_j represents the job completion time, the unit penalty represents the number of late jobs.

Applied Approaches

In this section we describe two *schedule-based* approaches we propose to solve the considered job scheduling problem. First, description of *Earliest Gap—Earliest Deadline First* policy used to create an initial schedule in an incremental fashion is given. Next, Tabu search algorithm is described, which periodically optimizes the initial solution according to the objective function. The schedule is represented as an array of particular machine's schedules, i.e., $schedule$:= $[mach\_sched_1, .., mach\_sched_m]$. Using this notation, schedule of machine i (i.e., $mach\_sched_i$) is expressed as $schedule[i]$ in the following text.

Earliest Gap—Earliest Deadline First

EG-EDF policy is used to add newly arrived job into the existing schedule, i.e., $schedule_{initial}$. This allows us to reuse existing solution so that the schedule is built incrementally over time which results in shorter algorithm runtime in comparison with re-computing the whole schedule from scratch. The policy follows the objective function by applying simple strategy (see Algorithm 1) which determines which particular machine from the set of all suitable machines will execute the job. When the EG-EDF policy finishes its execution it places the new job into this particular machine's schedule. Following algorithm is used. First, machines are sorted according to their R_i and s_i in decreasing order. Then all suitable machines are subsequently tested whether a suitable gap for the new job exists in their schedule. A gap is considered to be a period of idle CPU time. A new gap appears in the schedule every time the number of currently available CPUs (w.r.t. existing schedule) of the machine is lower than the number of CPUs requested by a job. Moreover if there are more gaps in a specific machine's schedule we always use the *earliest* one due to

Algorithm 1 Earliest Gap—Earliest Deadline First(job)

1: $schedule_{initial}$:= [$mach\_sched_1, .., mach\_sched_m$]; $schedule_{new} := \emptyset$; $schedule_{best} := \emptyset$; gap_found := **false**;
2: **for** $i := 0$ to m **do**
3: **if** $machine_i$ is suitable to perform job **then**
4: **if** suitable gap for job was found in $schedule_{new}[i]$ **then**
5: gap_found := **true**;
6: $schedule_{new}$:= $schedule_{initial}$;
7: $schedule_{new}[i]$:= place job into found gap in $schedule_{new}[i]$;
8: **else if** gap_found = **false then**
9: $schedule_{new}$:= $schedule_{initial}$;
10: $schedule_{new}[i]$:= place job into $schedule_{new}[i]$ using EDF strategy;
11: **end if**
12: **if** AcceptanceCriterion($schedule_{best}, schedule_{new}$) = **true then**
13: $schedule_{best}$:= $schedule_{new}$;
14: **end if**
15: **end if**
16: **end for**
17: **return** $schedule_{best}$

higher probability that the deadline of a job will be met. Even more importantly it may not be possible to place later jobs to earlier gaps as the time is running and machine may spend its time being idle not having a suitable job for the gap. If a suitable gap is found then the job is assigned to it and the new resulting schedule is evaluated according to $AcceptanceCriterion$ function w.r.t. the best so far found $schedule_{best}$. If such gap is not found in this machine's schedule and no gap was found so far on previously tested machines then the job is placed into the machine's schedule using EDF policy. Again the $AcceptanceCriterion$ is used to decide whether this solution, i.e., $schedule_{new}$ is better then the current $schedule_{best}$. If this is the case, then the $schedule_{new}$ becomes the new $schedule_{best}$ (see line 13). Once there was some gap found in some previously tested machine's schedule then only better gaps on remaining machines are searched and evaluated and EDF is never used again (see line 8). After all suitable machines were tested the $schedule_{best}$ is returned as the newly found solution.

$AcceptanceCriterion$ is used to decide whether $schedule_{new}$ is better than so far known best solution $schedule_{best}$ (see Algorithm 2). The decision is taken upon the value of $weight$ (see line 9) which is computed as a sum of $weight_{makespan}$ and $weight_{deadline}$. They express our two objectives—machine usage and deadlines respectively. When $weight_{makespan}$ is positive it means that $schedule_{new}$ has lower makespan than $schedule_{best}$ which means that also the machine usage will be better[1]. Similarly positive $weight_{deadline}$ value means that $schedule_{new}$ has lower number of delayed jobs ($U_{new} \leq U_{best}$). Obviously, some correction are needed when $makespan_{best}$ or $nondelayed_{best}$ are equal to zero but we do not present them to keep the code clear. Finally, the first line of $AcceptanceCriterion$ guarantees that the $schedule_{best}$ will be initialized correctly for the purposes of following iterations of EG-EDF (at least one $schedule_{best}$ will be always found).

[1]If there is a gap being filled with newly arriving job, then $weight_{makespan}$ is equal to zero because the new job "fits" within an existing gap and the makespan does not change. Although this situation increase machine usage it is not recognized by the $AcceptanceCriterion$ algorithm. This is the reason why we prefer gaps over EDF in EG-EDF policy.

Tabu Search

Although EG-EDF policy is trying to increase the machine usage and also to meet the job deadlines either by finding suitable gaps or through EDF policy, it only manipulates with the newly arriving job. Previously scheduled jobs are not considered by EG-EDF when building the new schedule. In such case many gaps in the schedule may remain which could be efficiently used by suitable jobs already present in the schedule. We apply a Tabu search (Glover and Laguna 1998) optimization algorithm which increases both machine usage and the number of jobs executed respecting their deadlines. It only manipulates jobs prepared for execution—jobs already running are not affected since the job preemption is not supported. In proposed solution we move "later" jobs from the end of some machine's schedule into earliest suitable gaps appearing in some machine's schedule. The idea behind this approach is twofold. First, the filling of *early gaps* helps to increase machine usage—otherwise this gap would soon result in insufficient machine usage. Second, jobs from the end of current schedule are *more likely to be delayed*, therefore it is reasonable to move them forward. Moreover, once such job is removed from its position remaining jobs in the schedule may often be executed earlier which also *increase the probability* that their deadline will be met.

Proposed solution is described in detail by Algorithm 3. In each iteration specific machine's schedule and one job from that schedule is selected as a candidate for move. The machine being selected is the one with highest number of delayed jobs. The job being selected is the last non-tabu job from this machine's schedule, i.e., $job \notin tabu_{jobs}$. Once the job is selected we remove it from its current position and we

Algorithm 2 AcceptanceCriterion($schedule_{best},\ schedule_{new}$)

if $schedule_{best} = \emptyset$ **then**
  return true;
end if
compute $makespan_{best}$ and $nondelayed_{best}$ according to $schedule_{best}$;
compute $makespan_{new}$ and $nondelayed_{new}$ according to $schedule_{new}$;
$weight_{makespan}$:= $(makespan_{best} - makespan_{new})/(makespan_{best})$;
$weight_{deadline}$:= $(nondelayed_{new} - nondelayed_{best})/(nondelayed_{best})$;
$weight$:= $weight_{makespan} + weight_{deadline}$;
if $weight > 0.0$ **then**
  return true;
else
  return false;
end if

Algorithm 3 Tabu Search($iterations$)

1: $schedule_{best}$:= $[mach\_sched_1, .., mach\_sched_m]$; $schedule_{new}$:= $schedule_{best}$; $tabu_{jobs}$:= $\emptyset$; $machines_{used}$:= $\emptyset$;
2: **for** $i := 0$ to $iterations$ **do**
3: $source$:= k such that: $k \in (1..m),\ machine_k \notin machines_{used},\ schedule_{new}[k]$ has highest number of delayed jobs;
4: **if** $source$ = $null$ **then**
5: $machines_{used}$:= $\emptyset$; (All machines were used – start a new round)
6: continue with new iteration;
7: **end if**
8: job := last job from $schedule_{new}[source]$ such that: $job \notin tabu_{jobs}$;
9: **if** job = $null$ **then**
10: $machines_{used}$:= $machines_{used} \cup machine_{source}$; (No non-tabu job is available in $schedule_{new}[source]$)
11: continue with new iteration;
12: **end if**
13: remove job from $schedule_{new}[source]$;
14: **if** MoveJob($job,\ schedule_{best},\ schedule_{new}$) = **false then**
15: $schedule_{new}$:= $schedule_{best}$; (returns job to the original position);
16: **else**
17: $schedule_{best}$:= $schedule_{new}$; (updates the best so far found solution)
18: **end if**
19: $tabu_{jobs}$:= $tabu_{jobs} \cup job$; (and remove oldest item if $tabu_{jobs}$ is full)
20: **end for**
21: **return** $schedule_{best}$

try to find a suitable gap where the job would fit in. This is performed by the $MoveJob$ function. First the set of all machines is randomly permuted and then all machines are subsequently tested in a loop whether a suitable gap exists in their schedules. If the gap is found the job is moved to it and the $AcceptanceCriterion$ is computed. If this move is accepted then $MoveJob$ returns true and the $schedule_{best}$ is updated with $schedule_{new}$ (see line 17). Otherwise the job is removed from the gap and next machine's schedule is investigated w.r.t. existence of suitable gap. This cycle continues until better schedule is found or all machines were examined. In case that no better solution was found then $MoveJob$ returns false and the job is returned to its original position (see line 15). Finally, the recent job is placed into $tabu_{jobs}$—so that it can not be chosen in the next few iterations—and a new iteration of Tabu search starts. If in some iteration the selected machine's schedule contains only tabu jobs it means that all of them were selected in few previous iterations, therefore we add this machine into $machines_{used}$ list so that it will not be chosen as a $source$ candidate in next iterations (see lines 10 and 3 respectively). When all machines are present in $machines_{used}$ it means that we went through all machine's schedules, therefore we clear the list and start another iteration (see line 5). This guarantees that all machines will become candidates if sufficient number of $iterations$ is given.

Experimental Evaluation

In order to verify the feasibility of the EG-EDF and Tabu search solutions number of experiments have been conducted. Since current Grid schedulers are mostly based on queues the evaluation was performed by comparing our solutions with common queue-based algorithms such as FCFS, EASY backfilling (EASY BF), and Flexible backfilling (Flex. BF). EASY backfilling (Skovira et al. 1996) is an optimization of the FCFS algorithm, which tries to maximize the machine usage. If the first queued job has to wait

Algorithm 4 MoveJob ($job,\ schedule_{best},\ schedule_{new}$)

```
1: Permute the list of machines to test them in random order;
2: for j := 0 to m do
3:    if machine_j is suitable to perform job and suitable gap for job was found in schedule_new[j] then
4:       schedule_new[j] := place job into found gap in schedule_new[j];
5:       if AcceptanceCriterion(schedule_best, schedule_new) = true then
6:          return true;
7:       else
8:          schedule_new[j] := schedule_best[j] (removes the proposed move);
9:       end if
10:   end if
11: end for
12: return false;
```

until necessary machine(s) become available then other jobs from the queue that may use available machines are scheduled immediately in case that they will not delay the first waiting job. While in FCFS such machines would be idle, in EASY backfilling they are "backfilled" with suitable jobs. The Flexible backfilling (Techiouba et al. 2008) is a modification of EASY backfilling where jobs are prioritized according to the scheduler goals and then queued according to their priority value, and selected for scheduling according to this priority order. In this case, the priority was computed respecting the proximity of job deadline, job waiting time and the job execution time. The priority of a job increases as its deadline is approaching, time spent in the queue is growing or its execution time is lower w.r.t. remaining queued jobs. Job priorities are updated at each job submission or completion event and then new scheduling round is started.

To simulate the dynamic Grid environment we used Alea Simulator (Klusáček, Matyska, and Rudová 2008), which is an extended version of the GridSim toolkit. Grid was made up of 150 machines with different CPU number and speed. Since current systems often do not support specific QoS related requirements such as use of job deadlines we were forced to create our own workload traces since we are not aware of any publicly available trace that would include job deadlines. Simulations used five different streams each containing 3000 synthetically generated jobs using negative exponential distribution with different inter-arrival times between jobs (Capannini et al. 2007). According to the job inter-arrival times a different workload is generated through a simulation. Smaller this time is, greater the system workload is. Inter-arrival times were chosen in a way that the available computational power of the machines is able to avoid the job queue increasing when it is fixed equal to 5 seconds. Each job and machine parameter was randomly generated according to an uniform distribution[2]. Both sequential and parallel jobs were simulated, however parallel jobs were always executed only on one machine with a sufficient number of CPUs. Each simulation was repeated 20 times with different job attributes to obtain reliable values. The experiments were performed on an Intel Pentium 4 2.6 GHz machine with 512 MB RAM.

To evaluate the quality of schedule computed by EG-EDF policy and Tabu search, we use different criteria: the percentage of jobs executed do not respecting their deadline, the percentage of machine usage, percentage of weighted machine usage, makespan, the average job slowdown, and the average algorithm runtime spent by the scheduler to create a scheduling decision.

Discussion

In Figure 1 the percentage of jobs executed not respecting their deadline is shown. As expected, when the job inter-arrival time increases, the number of late jobs decreases. Moreover, it can be seen that both EG-EDF policy and Tabu search produced much better solutions than Flexible backfilling, EASY backfilling or FCFS. Tabu search outperforms all the other algorithms. In particular, it obtains nearly the same results as EG-EDF policy when the system contention is low (job inter-arrival time equal to 5).

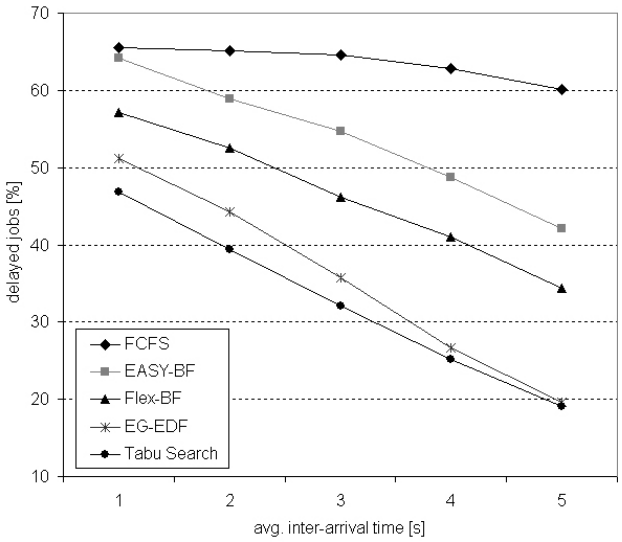

Figure 1: Average percentage of delayed jobs.

In Figure 2, the percentage of system usage is shown. Schedule-based algorithms are, in general, able to better ex-

[2]Following ranges were used: Job execution time [500–3000], jobs with deadlines 70%, number of CPUs required by job [1–8], number of CPUs per machine [1–16], machine speed [200–600]

ploit the system computational resources. However, when the system contention is low the solutions we propose obtained worse results concerning machine usage than the queue-based techniques. In such situation the schedule is empty for lots of machines which decreases machine usage. The reason for this behavior is that in such situation—using the schedule—our algorithms prefer putting waiting jobs rather onto fast machines because they will finish them earlier than if they were executed immediately but on a slow machine. Such situation will never occur for none of the queue-based algorithms since they make their decisions w.r.t. current situation and are not able to predict future behavior as the schedule-based solution does. Since we are dealing with heterogeneous machines more realistic results are those obtained by *weighted machine usage* criterion. Here the usage of $machine_i$ is computed as $machine\_usage_i \cdot s_i$ so the speed s_i of $machine_i$ is used as a weight. Therefore, weighted machine usage express the amount of utilized CPU operations. As discussed in (Tang and Chanson 2000) when a choice is to be made between two machines it is better to highly utilize fast machine rather than slow machine since fast machine computes much more operations in given time than slow machine. It is important to notice that such scenario is not covered by classic machine usage criterion where only the proportion of used and available CPUs is measured disregarding their speed. Figure 3 shows how schedule-based approach significantly outperforms remaining queue-based algorithms. We can see that similar effect is also clearly visible in Figure 4 representing the makespan. Since schedule-based methods are able to better utilize fast machines more jobs are completed in shorter time. Again, notice that this behavior is not recognizable in Figure 2.

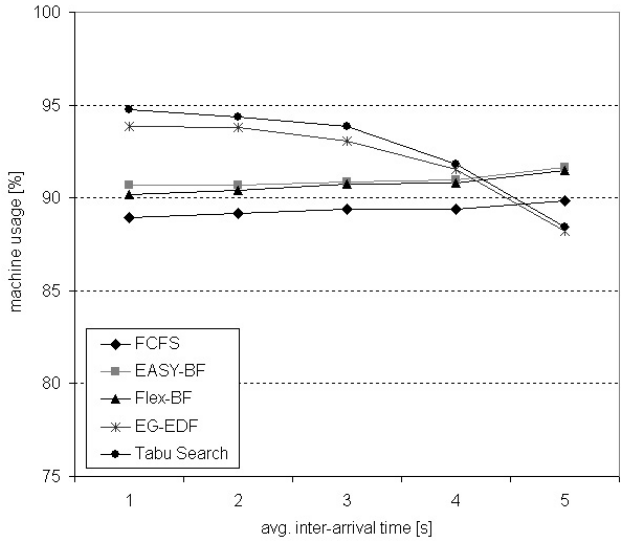

Figure 2: Machine usage.

Figure 5 shows the average execution time (runtime) spent by the scheduler to come up with scheduling decision for one job. It is computed by measuring the system CPU time

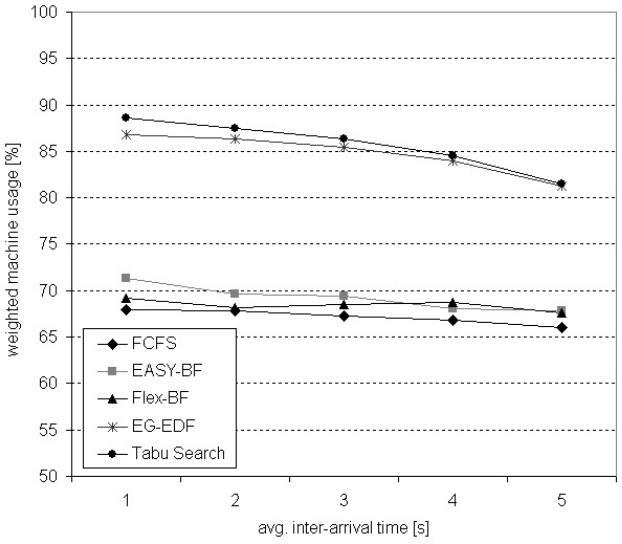

Figure 3: Weighted machine usage.

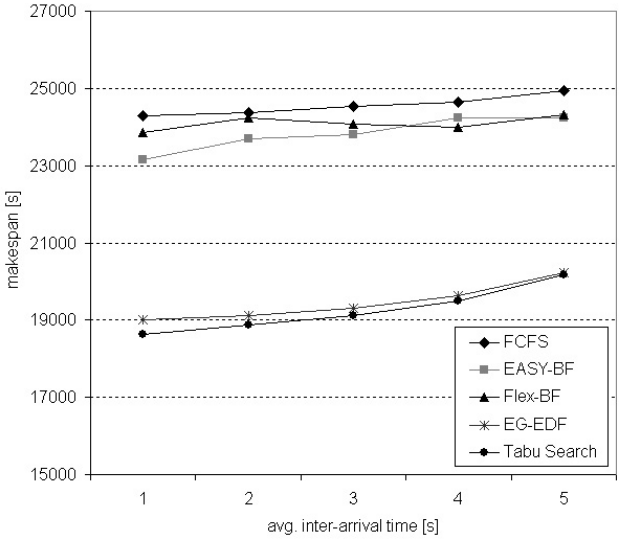

Figure 4: Makespan (compare proportion with Figure 3).

spent at each scheduling event. The runtime for FCFS is very low w.r.t. to EASY and Flexible backfilling for which it grows quickly as a function of the job queue length. Although the Flexible backfilling has to re-compute job priorities at each scheduling event, and then has to sort the queue accordingly, it causes minimal growth of its run time compared to the EASY backfilling thanks to the application of an efficient sorting algorithm.

Local search based algorithms are often considered to be very time consuming. Our implementation, which exploits an incremental approach based on the reuse of previously computed solution, is able to guarantee a shorter and stable execution time w.r.t. the other algorithms. In particular, EG-EDF policy is fast and it always generates acceptable sched-

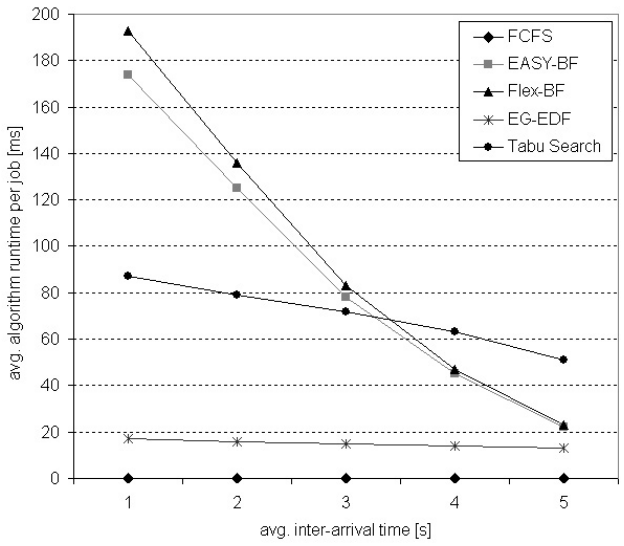

Figure 5: Average algorithm runtime per job.

ule, so we can stop Tabu search optimization at any time if prompt decisions are required, or even do not use it at all.

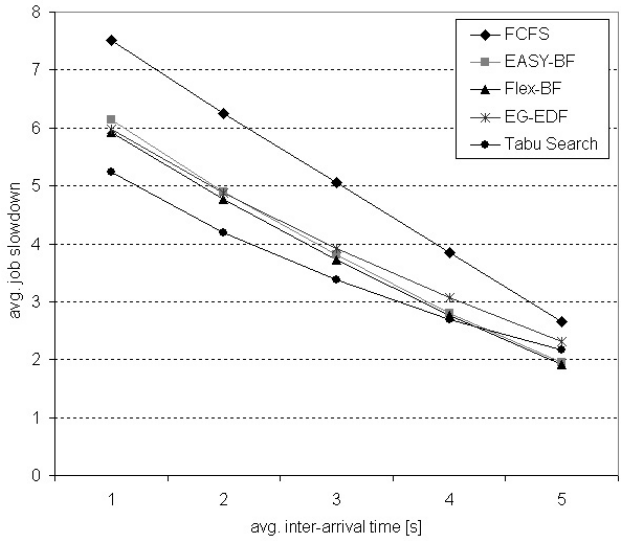

Figure 6: Average job slowdown.

Figure 6 shows the average job slowdown. It is computed as $(Tw+Te)/Te$, with Tw the time that a job spends waiting to start its execution, and Te the job execution time (Mu'alem and Feitelson 2001). This shows us how the system load delays the job execution. As expected, higher the system contention is, higher the job slowdown is. In this case the better results were obtained by Tabu search, which are nearly the same as those obtained by the Flexible backfilling algorithm. Unlike to Flexible backfilling, the slowdown was not explicitly considered neither in EG–EDF nor in the Tabu search, still the "backward to forward" gap-filling strategy was useful even in this case.

Conclusion and Future Work

We exploited schedule-based approaches to efficiently address Quality of Service requirements of Grid users towards processing of their jobs. Also, the overall Grid utilization was emphasized to address the Grid resource owners point of view. Schedule-based algorithms demonstrated significant improvement when decreasing the number of late jobs while keeping the machine usage high. This would not be possible without the application of effective gap-filling method. Tabu search algorithm proved to be more successful in decreasing the number of delayed jobs over Flexible backfilling—on the other hand precise job execution time was known in this case so the advantage of schedule-based solution took effect. The incremental approach used in the schedule-based solutions allowed to keep the algorithm runtime stable and low. From this point of view both EASY and Flexible backfilling are more time consuming since their runtime is growing with the size of the queue more quickly.

In the future we would like to extend our current model with network simulation and also include certain level of uncertainty such as job execution time estimations or dynamic resource changes. Next we will study their effect on the performance of schedule-based methods. Since we are interested in highly dynamic environments such as Grids, uncertainty and imprecision of information together with dynamic changes represent more realistic scenario. Usually, this is not a crucial issue for a lot of queue-based algorithms because they are designed to deal with dynamic changes and often require very limited amount of information at the cost of no or very limited guarantee of QoS. Here the schedule-based approach and generally every technique that aims to guarantee certain behavior—e.g., those using advanced reservation—relies on the precision of available information much more. Without that, the reliability of computed schedule is limited, thereby we are aware that some changes will have to be done when e.g., the job execution time estimates will not meet the real job execution time. In such situation local change or limited rescheduling will be necessary to keep the schedule up to date.

Acknowledgments

This work was kindly supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research intent No. 0021622419, by the Grant Agency of the Czech Republic with grant No. 201/07/0205, and by the EU CoreGRID NoE (FP6-004265).

References

Abraham, A.; Buyya, R.; and Nath, B. 2000. Nature's heuristics for scheduling jobs on computational Grids. In *The 8th IEEE International Conference on Advanced Computing and Communications (ADCOM 2000)*, 45–52.

Armentano, V. A., and Yamashita, D. S. 2000. Tabu search for scheduling on identical parallel machines to minimize mean tardiness. *Journal of Intelligent Manufacturing* 11:453–460.

Baraglia, R.; Ferrini, R.; and Ritrovato, P. 2005. A static mapping heuristics to map parallel applications

to heterogeneous computing systems: Research articles. *Concurrency and Computation: Practice and Experience* 17(13):1579–1605.

Capannini, G.; Baraglia, R.; Puppin, D.; Ricci, L.; and Pasquali, M. 2007. A job scheduling framework for large computing farms. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'07)*.

Cluster Resources. 2008. *Moab workload manager administrator's guide, version 5.1.0*. http://www.clusterresources.com/products/mwm/docs/.

Foster, I., and Kesselman, C. 1998. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann.

Gentzsch, W. 2001. Sun Grid Engine: towards creating a compute power grid. In *Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid*, 35–36.

Glover, F. W., and Kochenberger, G. A., eds. 2003. *Handbook of metaheuristics*. Kluwer.

Glover, F. W., and Laguna, M. 1998. *Tabu search*. Kluwer Academic Publishers.

Hovestadt, M.; Kao, O.; Keller, A.; and Streit, A. 2003. Scheduling in HPC resource management systems: Queuing vs. planning. In Feitelson, D. G.; Rudolph, L.; and Schwiegelshohn, U., eds., *9th International Workshop, JSSPP 2003*, volume 2862 of *LNCS*, 1–20. Springer.

Huedo, E.; Montero, R.; and Llorente, I. 2005. The GridWay framework for adaptive scheduling and execution on Grids. *Scalable Computing: Practice and Experience* 6(3):1–8.

Jones, J. P. 2005. *PBS Professional 7, administrator guide*. Altair.

Klusáček, D.; Rudová, H.; Baraglia, R.; Pasquali, M.; and Capannini, G. 2008. Comparison of multi-criteria scheduling techniques. In *Integrated Research in Grid Computing*. Springer. To appear.

Klusáček, D.; Matyska, L.; and Rudová, H. 2008. Alea – Grid scheduling simulation environment. In *7th International Conference on Parallel Processing and Applied Mathematics (PPAM 2007)*, volume 4967 of *LNCS*, 1029–1038. Springer.

Mu'alem, A. W., and Feitelson, D. G. 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems* 12(6):529–543.

Pinedo, M. 2005. *Planning and scheduling in manufacturing and services*. Springer.

Skovira, J.; Chan, W.; Zhou, H.; and Lifka, D. A. 1996. The EASY - LoadLeveler API Project. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 41–47. Springer.

Smith, W.; Foster, I.; and Taylor, V. 2000. Scheduling with advanced reservations. In *International Parallel and Distributed Processing Symposium (IPDPS '00)*.

Stucky, K.-U.; Jakob, W.; Quinte, A.; and Süß, W. 2006. Solving scheduling problems in Grid resource management using an evolutionary algorithm. In Meersman, R., and Tari, Z., eds., *OTM Conferences (2)*, volume 4276 of *LNCS*, 1252–1262. Springer.

Subrata, R.; Zomaya, A. Y.; and Landfeldt, B. 2007. Artificial life techniques for load balancing in computational Grids. *Journal of Computer and System Sciences* 73(8):1176–1190.

Süß, W.; Jakob, W.; Quinte, A.; and Stucky, K.-U. 2005. GORBA: A global optimising resource broker embedded in a Grid resource management system. In Zheng, S. Q., ed., *International Conference on Parallel and Distributed Computing Systems, PDCS 2005*, 19–24. IASTED/ACTA Press.

Tang, X., and Chanson, S. T. 2000. Optimizing static job scheduling in a network of heterogeneous computers. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, 373 – 382. USA: IEEE Computer Society.

Techiouba, A. D.; Capannini, G.; Baraglia, R.; Puppin, D.; and Pasquali, M. 2008. Backfilling strategies for scheduling streams of jobs on computational farms. In Danelutto, M., and Getov, V., eds., *Making Grids Work*. USA: Springer. ISBN 978-0-387-78447-2.

Thain, D.; Tannenbaum, T.; and Livny, M. 2005. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience* 17(2-4):323–356.

Xu, M. Q. 2001. Effective metacomputing using LSF multicluster. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, 100–105. IEEE Computer Society.