Efficient Grid Scheduling through the Incremental Schedule-based Approach

Dalibor Klusáček and Hana Rudová

Faculty of Informatics, Masaryk University

Botanická 68a, Brno 602 00

Czech Republic

phone: +420 549 491 893

{xklusac, hanka}@fi.muni.cz

Abstract

While Grid users demand good performance for their jobs, this requirement is often not satisfied by the widely used queue-based scheduling approaches. This paper concentrates on the application of schedule-based methods that improve on both the service delivered to the user and the traditional objective of machine usage. Importantly, the interaction between the incremental application of these methods and the dynamic character of the problem allows reasonable runtimes to be achieved. Two new schedule-based methods designed to schedule dynamically arriving jobs on machines in a computational Grid are formally described in the paper. The Earliest Gap — Earlier Deadline First (EG-EDF) policy fills the earliest gaps (EG) in the known schedule with newly arriving jobs, incrementally building a new schedule. If the gap is not suitable for an incoming job, the EDF policy is used to modify the existing schedule. A Tabu search algorithm is used to further optimize the schedule by again moving selected jobs into the earliest suitable gaps. The proposed incremental schedule-based methods are compared with some of the most common queue-based scheduling algorithms such as FCFS (First Come First Served), EASY backfilling (Extensible Argonne Scheduler sYstem), Flexible backfilling, and with the non-incremental version of the EG-EDF schedule-based policy.

Keywords: Grid, Scheduling, Policy, Tabu search, Heuristic, Incremental approach

1 Introduction

The purpose of the *Grid technology* is to manage large and heterogeneous computer environment allowing various users to easily access the Grid resources, submit their jobs into the system and guaranteeing them *nontrivial Quality of Service (QoS)* while hiding the complexity of the system itself by providing powerful but simple interfaces (Foster and Kesselman, 1998). Moreover, not only users but also the resource owners should be satisfied. Typically, the resource owners prefer to keep the overall resource usage reasonably high. Therefore, *multi-objective criteria* have to be met. In order to meet these goals sophisticated and automated scheduling techniques should be applied. On the other hand, Grid is highly dynamic, distributed, and heterogeneous environment therefore the scheduling is an extremely difficult task if good performance, QoS or robustness is required.

Current scheduling techniques applied in the Grids are mostly based on the queueing systems of various types which are designed with respect to specific needs of the Grid technology. Present systems like PBS (Jones, 2005), LSF (Xu, 2001), Sun Grid Engine (Gentzsch, 2001), Condor (Thain et al., 2005) together with complex Grid management systems such as GridWay (Huedo et al., 2005) or Moab (Clu, 2008) represent de facto standard solutions. Still, they are using queue-based scheduling policies whose extension towards solving more complex problems is not easy or it is even impossible. While single objectives can often be optimized with a proper queue-based policy, complex objectives including, e.g., response time, flow time, slowdown, deadlines, resource utilisation, etc., are hard to handle by a queue-based solution, especially for the common users. Nowadays, users are often forced to cheat when looking for a good performance for 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 (the booking of certain time and resources for a job in advance) is often implemented to guarantee certain QoS level, however this functionality is often restricted by the 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 not having reservation often appears (Smith et al., 2000). The goal of our work is to support both the users' and Grid administrators' requirements. While the advanced reservation is a very complex issue, in this paper the needs of the users are expressed by a simpler objective function focusing on maximizing the number of jobs that meet their deadline (Capannini et al., 2007; Abramson et al., 2005) and the needs of the system administrators are expressed by the machine usage criterion, commonly used in the real Grids.

In dynamic environments such as Grids, resources may change, jobs are not known in advance and they appear while others are running. *Schedule-based approach* allows precise mapping of jobs to machines

in time. This enables to apply advanced scheduling algorithms (Pinedo, 2005) such as local search-based methods (Hoos and Stützle, 2005; Glover and Kochenberger, 2003) to optimize the schedule. Due to their computational cost, these approaches were mostly applied to the static problems, assuming that all the jobs and resources are known in advance which allows to create the schedule for all jobs at once (Armentano and Yamashita, 2000; Baraglia et al., 2005). CCS (Hovestadt et al., 2003) as well as GORBA (Süß et al., 2005) are both advanced Grid resource management systems that use schedule instead of the queue(s) to schedule workflows (GORBA), or sequential and parallel jobs while supporting the advanced reservation (CCS). GORBA uses simple policies for schedule creation and an evolutionary algorithm for its optimization 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. Although it helps to keep the schedule up to date, for large number of jobs this approach may be quite time consuming as was discussed in the case of GORBA (Stucky et al., 2006). Several papers (Abraham et al., 2000; Subrata et al., 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 et al., 2000), and (Subrata et al., 2007) does include resource changes but no dynamic job arrivals.

In this paper, a formalized description of our schedule-based solution is provided. A brief description of some of the ideas used was already given in our previous work (Klusáček et al., 2008b). Our approach allows to efficiently schedule *dynamically arriving* jobs onto the machines of the computational Grid. The initial schedule is generated by a simple and fast EG-EDF policy and then periodically optimized with the Tabu search (Glover and Laguna, 1998) algorithm. In comparison with other approaches (Hovestadt et al., 2003; Süß et al., 2005), here the *policy* and the *local search* are used in an *incremental* fashion. It means that last computed schedule is used as the starting point for building a new, up to date schedule. This leads to a reasonable computational cost since the schedule is not rebuilt from scratch. Moreover, an inclusion of the multi-criteria objectives is allowed which focuses on providing a good performance both to the Grid users (deadlines) and the Grid administrators (machine usage). The success of the solution is based on an efficient method — inspired by the backfilling approach commonly used in the queue-based systems (Shmueli and Feitelson, 2003) — which detects and fills existing gaps in the schedule with suitable jobs. It allows to increase both the number of non-delayed jobs and the machine usage by limiting the fragmentation of the processor time. Our solution was evaluated through a set of experiments against typical queue-based scheduling algorithms like FCFS, EASY backfilling (Skovira et al., 1996) and Flexible backfilling (Techiouba et al., 2008). Also for the purpose of evaluation, the non-incremental version of the EG-EDF was developed to simulate the "re-compute from from scratch" approach, which is used in the CCS or GORBA.

2 Problem Description

In our study a fixed set of m machines is expected and changes in the set of jobs are allowed. As the time is running, new jobs may appear and processing of other jobs is completed meanwhile. Newly arriving jobs are placed into the schedule which define where and when they will be executed. There are n jobs in total. Each job is characterized by 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 a known processing time $p_{i,j}$ which depends on the CPU speed of the machine i. Job also requires R_j number of CPUs for its execution ($R_j > 0$). Finally, C_j is the job completion time. Resources are computational machines with known capacity R_i, representing the number of CPUs. All CPUs within one machine have the same speed s_i representing the number of operations per second. Different machines may have different speed and different number of CPUs. All machines use the Space Sharing processor allocation policy which allows the parallel execution of k jobs on the machine i if $R_i \geq \sum_{j=1}^{k} R_j$.

Various objective functions can be considered such as the makespan ($C_{\max} = \max\limits_{1 \leq j \leq n} C_j$), or the average response time ($\frac{1}{n} \sum_{j=1}^{n} (C_j - r_j)$). Our scheduler aims to maximize both the *machine usage* and the number of *non-delayed jobs*. 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. Momentary machine usage at the given time t is computed as $u_t = \sum_{i=1}^{m} R_i^t \ / \ \sum_{i=1}^{m} R_i$, where R_i^t is the number of busy CPUs on the machine i at the time t. The average machine usage is computed as $u_{avg} = \frac{1}{C_{\max}} \sum_{t=0}^{C_{\max}} u_t$. Since heterogeneous machines are considered, the *average weighted machine usage* criterion (wu_{avg}) is used in this paper. In this case the momentary weighted usage is computed as $wu_t = \sum_{i=1}^{m} R_i^t \cdot s_i \ / \ \sum_{i=1}^{m} R_i \cdot s_i$ where the speed s_i of the machine i is used as a weight. Therefore, $wu_{avg} = \frac{1}{C_{\max}} \sum_{t=0}^{C_{\max}} wu_t$ expresses the average 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 the fast machine rather than the slow machine since the fast machine computes much more operations in the given time than the slow one. It is important to notice that such a scenario is not covered by the classic machine usage criterion where only the proportion of used and available CPUs is measured disregarding their speeds.

The number of delayed jobs is represented by the *unit penalty* function $U = \sum_{j=1}^{n} U_j$ where $U_j = 1$ if the job j is *delayed*, i.e., $C_j > d_j$. Otherwise U_j is equal to 0. An additional criterion of the *average slowdown* is considered to study delays of all jobs in the system which is an important performance measure for the jobs with no deadline. The average slowdown is computed as $\frac{1}{n} \sum_{j=1}^{n} \frac{C_j - r_j}{p_{i,j}}$ for the job j processed on the machine i.

3 Applied Approaches

In this section, the two proposed *schedule-based* approaches for solving of the considered job scheduling problem are described. First, the incremental *Earliest Gap — Earlier Deadline First* policy used to construct the initial schedule is presented. Next, the Tabu search algorithm, which periodically optimizes the initial solution according to the objective function, is presented. The schedule is represented as an array of particular machine's schedules, i.e., $schedule := [mach\_sched_1, .., mach\_sched_m]$. Using this notation, the schedule of machine i (i.e., $mach\_sched_i$) is denoted as $schedule[i]$ in the following text. Single machine's schedule is stored as a linear list of jobs. This list is ordered according to the start times of the jobs. If two or more jobs in the machine's schedule have the same start time, then the one being assigned to the CPU with the smallest *id* becomes the predecessor of the remaining jobs in this list and so on. Moreover, once the $schedule[i]$ is built, also a check is performed whether a *gap* appears next to any job. A gap is considered to be the period of idle CPU time. It appears every time the number of available CPUs of the machine (regarding the existing schedule) is greater than the number of CPUs requested by the job(s) in the given time period. The gap also appears at the end of each machine's schedule (after the last job) or when there are no jobs in some machine's schedule at all. Gaps for the specific machine's schedule are stored within the same data structure as jobs — in the linear list $schedule[i]$.

3.1 Earliest Gap — Earlier Deadline First

The EG-EDF policy is used to add newly arrived *job* into the existing schedule, denoted as the $schedule_{initial}$. This allows to reuse existing solution and to build the schedule incrementally over the time which results in a shorter algorithm runtime in comparison with the re-computation of the complete schedule from scratch. The EG-EDF policy is implemented by Algorithm 1 that determines which particular machine from the set of all suitable machines will execute the *job* at what time. When the policy finishes its execution, it places the new *job* into the selected machine's schedule. The details of the implementation follows. All suitable machines are subsequently considered and a *suitable gap* for the new *job* is found in their schedule (line 5). The machine i is considered as *suitable* if it has enough CPUs to execute the *job*, i.e., $R_i \geq R_{job}$. Similarly, the gap in the schedule of machine i is suitable for the *job* if the gap's length (duration) is greater or equals to $p_{i,job}$ and the gap's size (number of free CPUs) is greater or equals to R_{job}. If there are more suitable gaps in the specific machine's schedule the *earliest* one is always used, due to the higher probability that the job's deadline will be met (*Earliest Gap (EG)* policy)[1]. More importantly, it may not be possible to place future jobs into these earlier gaps as the time is running and the machine may spend its time being idle not having

[1]If the machine i is suitable for the *job* then at least one suitable gap is found since there is always a gap situated at the end of machine's schedule, its length is infinite and its size is equal to R_i.

Algorithm 1 Earliest Gap — Earlier Deadline First(job)

1: $schedule_{initial}$:= [$mach\_sched_1, .., mach\_sched_m$]; $schedule_{new} := \emptyset$; $schedule_{best} := \emptyset$; $k := 0$; $gap := \emptyset$;
2: **for** $i := 0$ to m **do**
3: **if** $machine_i$ is suitable to perform job **then**
4: $schedule_{new} := schedule_{initial}$;
5: gap := find the earliest suitable gap for the job;
6: $schedule_{new}[i]$:= place job into the gap in the $schedule_{new}[i]$; (EG strategy)
7: **if** AcceptanceCriterion($schedule_{best}, schedule_{new}$) = **true then**
8: $schedule_{best} := schedule_{new}$;
9: **else**
10: $schedule_{new} := schedule_{initial}$;
11: k := index of the first $job_k \in schedule_{new}[i]$ whose $d_{job_k} > d_{job}$; (k is the index of the first job with a later deadline)
12: $schedule_{new}[i]$:= insert job into $schedule_{new}[i]$ between job_{k-1} and job_k; (EDF strategy)
13: **if** AcceptanceCriterion($schedule_{best}, schedule_{new}$) = **true then**
14: $schedule_{best} := schedule_{new}$;
15: **end if**
16: **end if**
17: **end if**
18: **end for**
19: **return** $schedule_{best}$

a suitable job for the gap. If the suitable gap is found then the job is placed to it and the new resulting schedule is evaluated according to the *AcceptanceCriterion* function with respect to the best so far found $schedule_{best}$. If this assignment is accepted, then the $schedule_{new}$ becomes the new $schedule_{best}$ (line 8). Otherwise this assignment is rejected, the job is removed from the gap and the second policy — *Earlier Deadline First (EDF)* — is applied. Our implementation of the EDF policy subsequently goes through the list of jobs in the $schedule_{new}[i]$ (schedule of machine i) and finds the first job_k such that $d_{job_k} > d_{job}$ holds. Incoming job is placed between job_{k-1} and job_k, shifting job_k and all later jobs in the machine's schedule. Note that not all the jobs in the $schedule_{new}[i]$ have to be ordered by their deadline — some job(s) having arrived earlier could have been assigned to this machine's schedule using the "gap-filling" EG policy, which does not consider deadline order at all. The newly constructed $schedule_{new}$ is again analyzed by the *AcceptanceCriterion* to decide whether this solution will be accepted as the new $schedule_{best}$ (line 14). Once all suitable machines have been tested the $schedule_{best}$ is returned as the newly found solution.

AcceptanceCriterion is used to decide whether the $schedule_{new}$ is better than the best so far known solution $schedule_{best}$ (see Algorithm 2). The decision is taken upon the value of the $weight$ (line 9), which is computed as the sum of the $weight_{usage}$ and the $weight_{deadline}$. They express our two objectives, the machine usage and the number of non-delayed job respectively. When the $weight_{usage}$ is positive it means that the $schedule_{new}$ provides higher machine usage than the $schedule_{best}$. Similarly, the positive $weight_{deadline}$ value means that the $schedule_{new}$ has lower number of delayed jobs ($U_{new} \leq U_{best}$). Obviously, some correction

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

1: **if** $schedule_{best} = \emptyset$ **then**
2: **return true**;
3: **end if**
4: compute $usage_{best}$ and $nondelayed_{best}$ according to the $schedule_{best}$;
5: compute $usage_{new}$ and $nondelayed_{new}$ according to the $schedule_{new}$;
6: $weight_{usage} := (usage_{new} - usage_{best})/(usage_{best})$;
7: $weight_{deadline} := (nondelayed_{new} - nondelayed_{best})/(nondelayed_{best})$;
8: $weight := weight_{usage} + weight_{deadline}$;
9: **if** $weight > 0$ **then**
10: **return true**;
11: **else**
12: **return false**;
13: **end if**

is needed when the $usage_{best}$ or the $nondelayed_{best}$ is equal to zero but it is not presented to keep the code clear. Finally, the first line of the *AcceptanceCriterion* guarantees that at least one $schedule_{best}$ will be found by the EG-EDF.

3.2 Tabu Search

Although the EG-EDF policy tries to increase the machine usage and to meet the job deadlines, either by finding suitable gaps or through the EDF policy, it only focuses on the newly arriving job. Previously scheduled jobs are not primarily considered by the EG-EDF when building a new schedule. In such case many gaps in the schedule may remain. These could be efficiently used by suitable jobs already present in the schedule. Therefore, the Tabu search optimization algorithm is applied which increases both the machine usage and the number of non-delayed jobs. Both delayed and non-delayed jobs are considered as candidates otherwise the diversity of the neighborhood drops down together with the quality of the solution as was observed during the tests. In the proposed solution "later" jobs — starting with the last job in the chosen machine's schedule and continuing to its front — are moved into the earliest suitable gaps appearing in some machine's schedule. The idea behind this approach is twofold. First, the filling of the *early gaps* helps to increase the machine usage, otherwise the gap would soon result in an insufficient machine utilization. Second, chosen jobs are *more likely to be delayed* since they were located in the "later" parts of the schedule, therefore it is reasonable to move them forward. Moreover, once such job is removed from its position, remaining jobs in the schedule may often start their execution earlier (they are shifted to the earlier start times) which *increases the probability* that their deadlines will be met.

Algorithm 3 describes the proposed solution in detail. In each iteration a specific machine is selected and one job from its schedule is chosen as a candidate for the move. The machine being selected is the one with the highest number of delayed jobs. The job being selected is the last non-tabu job from this

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 the highest number of delayed jobs;
4: **if** $source = null$ **then**
5: $machines_{used} := \emptyset$; (all machines were used therefore clear the list and start a new round)
6: **continue**;
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 found in $schedule_{new}[source]$)
11: **continue**;
12: **end if**
13: remove job from $schedule_{new}[source]$;
14: **if** MoveJob(job, $schedule_{best}$, $schedule_{new}$) = **true then**
15: $schedule_{best} := schedule_{new}$; (updates the best so far found solution)
16: **else**
17: $schedule_{new} := schedule_{best}$; (places job to the original position);
18: **end if**
19: $tabu_{jobs} := tabu_{jobs} \cup job$; (removes oldest item if $tabu_{jobs}$ is full)
20: **end for**
21: **return** $schedule_{best}$

machine's schedule, i.e., $job \notin tabu_{jobs}$. Once the job is selected, it is removed from its current position and an attempt to find a better $schedule_{new}$ — increasing the value of the objective function — is made by the *MoveJob* function (see Algorithm 4). If this attempt is successful the *MoveJob* returns true and the $schedule_{best}$ is updated with the $schedule_{new}$ (line 15). Otherwise the job is returned to its original position (line 17). Finally, the job is placed into the $tabu_{jobs}$ (line 19) — so that it cannot be chosen in next few iterations — and a new iteration of the Tabu search starts. If the selected machine's schedule contains only tabu jobs in some iteration, it means that all of them were selected in the previous few iterations, therefore this machine is added into the $machines_{used}$ list, so that it will not be chosen as the *source* candidate in the next iterations (lines 10 and 3 respectively). When all machines are present in the $machines_{used}$ it means that all machines' schedules were explored, therefore the list is cleared and another iteration starts (line 5). It guarantees that all machines will become candidates if a sufficient number of *iterations* is given.

The pseudo-code of the *MoveJob* function is displayed in Algorithm 4. It tries to find a better assignment for the *job* being moved. First, the set of all machines is randomly permuted and then the earliest suitable gap is found (line 4) at every suitable machine (line 3). Once it is found, the job is moved into the gap and the *AcceptanceCriterion* is computed (line 6). If it returns true then this move is accepted and *MoveJob* returns true (line 7). Otherwise, the job is removed from the gap (line 9) and the next machine is investigated. The cycle continues until a better schedule is found (line 7) or until all machines are examined (line 12), meaning

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

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

that no better solution has been found. In this case the $MoveJob$ returns false (line 13).

4 Experimental Evaluation

In order to verify the feasibility of the incremental EG-EDF (incr-EG-EDF) policy and the Tabu search (incr-EG-EDF+TS) optimization procedure, a number of experiments have been conducted. The evaluation was performed by comparing our solutions with some common queue-based algorithms such as the FCFS, the EASY backfilling (EASY-BF), and the Flexible backfilling (Flex-BF). The 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 until the necessary machine(s) become available, then other jobs from the queue that may use available machines are scheduled immediately, if they will not delay the first waiting job. While such machines would be idle in the FCFS, they are "backfilled" with suitable jobs in the EASY backfilling. The Flexible backfilling (Techiouba et al., 2008) is a modification of the EASY backfilling where jobs are prioritized based on the scheduler's goals, queued according to their priority and then selected for scheduling in this priority-based order. The priority is computed respecting the values of the job deadline, the job waiting time and the job execution time. Shorter execution time, closer deadline or longer waiting time among other queued jobs increase the resulting job priority. Job priorities are updated at each job submission or completion event and then the new scheduling round is started.

Moreover, the proposed incremental incr-EG-EDF policy has been modified to follow the "re-compute from scratch" approach (re-comp-EG-EDF) applied in the schedule-based systems such as CCS or GORBA. The re-comp-EG-EDF works in the following fashion. Every time a new job arrives the existing schedule is removed, the jobs are stored in a list and ordered according to their deadlines using the Earliest Deadline First policy. Then, the incr-EG-EDF is repeatedly applied, taking one job at a time and placing it into the

newly constructed schedule. It starts with the first job in the list (the one having the minimal deadline) and continues until all jobs are placed into the schedule. The non-incremental re-comp-EG-EDF has been used to show the available speedup of the incremental approach. Also, it helps to analyze whether the application of the incremental approach causes any significant inefficiency concerning the value of the objective function.

The dynamic Grid environment has been simulated by the Alea simulator (Klusáček et al., 2008a), which is an extended version of the GridSim toolkit (Buyya and Murshed, 2002). The Grid consists of 150 machines with different CPU number and speed. The workload traces for the experiments has been generated synthetically[2] since the current systems often do not support specific QoS-related features such as the use of job deadlines and — as far as we know — there are no publicly available traces that would include them. The simulations used five different streams, each containing 3000 synthetically generated jobs using negative exponential distribution with different inter-arrival times between the jobs (Capannini et al., 2007). According to the job inter-arrival times a different workload is generated through a simulation. The smaller this time is, the greater the system workload is. The inter-arrival times have been chosen in a way that the available computational power of the machines is able to keep the system workload stable when it is fixed equal to 5 seconds. Each job and each machine parameter was randomly generated according to a uniform distribution. Both sequential and parallel jobs were simulated. 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 QuadCore 2.6 GHz machine with 4096 MB RAM.

To evaluate the quality of the schedule computed by the incr-EG-EDF policy and the Tabu search, different criteria have been used: the number of delayed jobs, the percentage of machine usage, the makespan, the average job slowdown and the average algorithm runtime spent by the scheduler to create a scheduling decision.

4.1 Discussion

In the proposed solution the major part is related to the application of the Tabu search (TS) algorithm that periodically optimizes the initial solution created by the incr-EG-EDF. Tabu search performance relies on two main parameters which are discussed before proceeding to the final comparison. Clearly the average runtime of the TS depends on the number of *iterations* (see Algorithm 3) and also on the TS execution frequency, representing how many job arrivals appear between two TS executions. Several experiments which varied both by means of frequency and iterations were performed to determine the suitable setup for

[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]. All sources as well as workloads are available at: `http://www.fi.muni.cz/~xklusac/alea/`

the experimental problem. First, the influence of TS frequency was examined. In this case the total sum of iterations per experiment was always equal but the frequency differed. The frequencies were following: 2 jobs, 5 jobs, 10 jobs, 15 jobs and 20 jobs. Since the total sum of TS iterations per experiment was equal to 300 000 and there were 3000 job arrivals, the resulting distribution of iterations was following: 2:200, 5:500, 10:1000, 15:1500 and 20:2000. In this XX:YYY notation the XX represents the frequency and YYY represents the number of iterations of one TS. In Figure 1 (left), the influence of TS frequency on the required runtime is visible. Clearly, the more often the TS (2:200) is executed the higher the runtime is, due to the

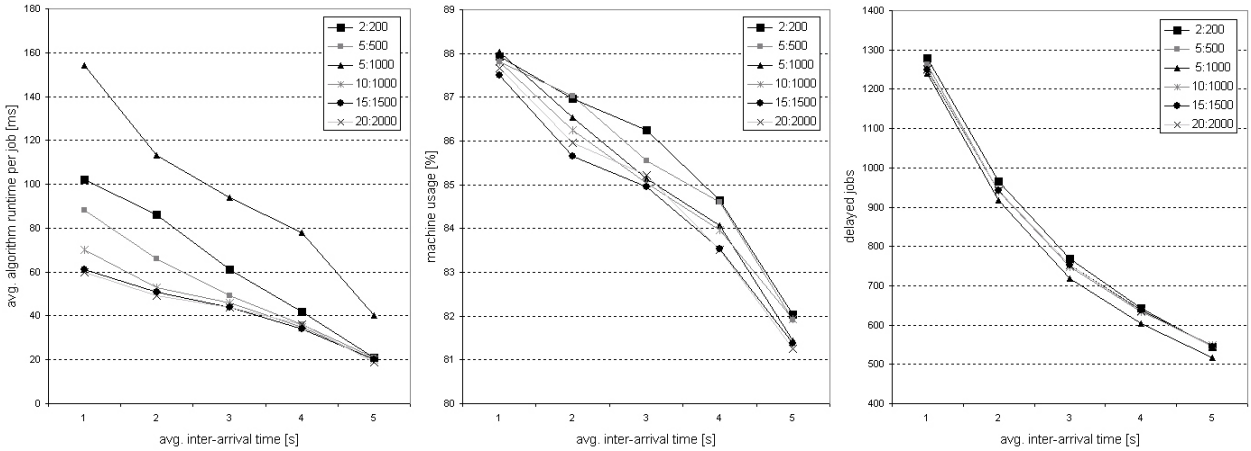

Figure 1: The influence of the TS iterations and the TS execution frequency on the algorithm runtime (left), the machine usage (middle) and the number of delayed jobs (right).

necessary setup times required to run the TS. On the other hand, when the TS is executed rarely (20:2000) the required runtime is the smallest one since the necessary setup time is amortized. Concerning the machine usage Figure 1 (middle) shows that the differences are very small in the order of one percent unit while the number of delayed jobs (right) is almost identical.

In the next step the influence of iterations was measured. The combination 5:500 was selected and its iterations were doubled which resulted in the experiment depicted as 5:1000. Figure 1 (left) shows that 5:1000 needs approximately twice the runtime needed by the 5:500 (the exact average value is equal to 1.9) which goes along with the expectations. Figure 1 (right) shows that 5:1000 is able to decrease the number of delayed jobs with respect to the remaining setups, however the improvement is rather small. Finally, Figure 1 (middle) shows, that the resulting machine usage is almost similar to the other setups. Using these experiences, the 5:500 TS setup has been used in the remaining experiments since it represents a reasonable trade-off between the optimization and the runtime performance.

Figure 2 (left) shows the number of jobs that failed to meet their deadline. As expected, when the job inter-arrival time increases, the number of delayed jobs decreases. Moreover, it can be seen that both the

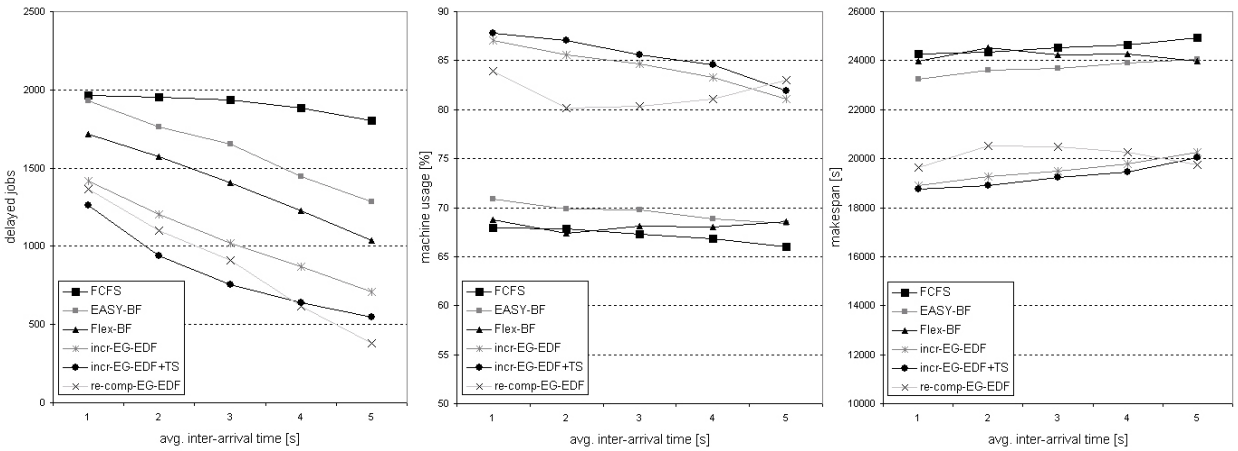

Figure 2: Number of delayed jobs (left), the machine usage (middle) and the makespan (right).

incr-EG-EDF and re-comp-EG-EDF policy produced much better solutions than the Flexible backfilling, the EASY backfilling or the FCFS. When the Tabu search is also applied (incr-EG-EDF+TS) — as a periodical optimization of the schedule generated by the incr-EG-EDF — it outperforms all the other algorithms in most cases and produces good results with respect to the re-comp-EG-EDF. In Figure 2 (middle), the percentage of machine usage is presented. It shows that the schedule-based approach significantly outperforms the remaining queue-based algorithms. Using the schedule, our algorithms prefer to wait for a while until the fast machine is available, finishing the job earlier than if executed immediately on a slow machine. Such situation will never happen for any of the used queue-based algorithms since they make their decisions according to the current situation. In contrast to the schedule-based solution they are unable to predict the future system behavior. Similar results are also clearly visible in Figure 2 (right) representing the makespan. Since the schedule-based methods are able to highly utilize the fast machines, more jobs are completed in a shorter time.

Figure 3 (left) shows the average execution time (runtime) spent by the scheduler to come up with the scheduling decision for one job. The runtime is in the logarithmic scale. It is computed by measuring the average system CPU time spent at each scheduling event. The runtime for FCFS is very low with respect to the EASY and Flexible backfilling for which the runtime grows as a function of the job queue length. Although the Flexible backfilling has to re-compute job priorities at each scheduling event and to sort the queue accordingly, it causes minimal growth of its runtime compared to the EASY backfilling thanks to the application of an efficient sorting algorithm. The incr-EG-EDF policy requires reasonably short runtime due to the application of the incremental approach. On the other hand the non-incremental re-comp-EG-EDF has the worst runtime of all algorithms. Moreover, the first two cases require more time than there is available — notice that the average inter-arrival time is 1 and 2 seconds while the algorithm requires 6.6

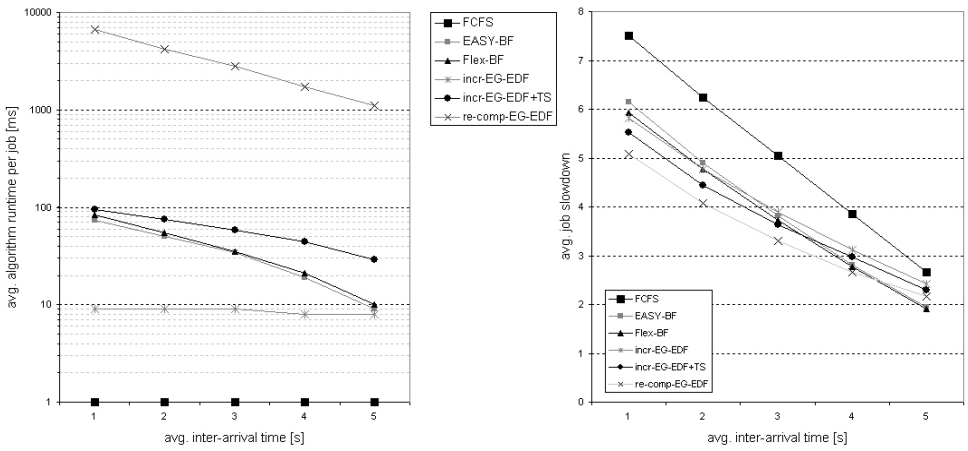

Figure 3: The average algorithm runtime per one job (left) and the average job slowdown (right).

and 4.2 seconds, respectively. Although it is possible to simulate such behavior, re-comp-EG-EDF policy is not applicable in the real world at all. Figure 3 (right) presents the average job slowdown. It shows how the system load delays the job execution. As expected, the higher the system contention is, the higher the job slowdown is. In this case better results were obtained by the re-comp-EG-EDF and incr-EG-EDF+TS solutions, which are nearly the same as those obtained by the Flexible backfilling algorithm. Unlike the Flexible backfilling, the slowdown was not explicitly considered neither in re-comp/incr-EG-EDF nor in the incr-EG-EDF+TS. Still, the "backward to forward" gap-filling strategy was useful even in this case.

5 Conclusion and Future Work

The incremental schedule-based approaches were exploited to efficiently address the QoS related requirements of the Grid users towards the processing of their jobs. Also the overall Grid utilization was emphasized to address the Grid resource owners' point of view. The schedule-based algorithms demonstrated significant improvement when decreasing the number of delayed jobs and maximizing the machine usage while keeping very reasonable runtime. As it was shown through the experimental evaluation, this would not be possible without the application of the incremental approach together with the efficient gap-filling method. Both the EG-EDF and the Tabu search proved to be more successful over the FCFS, EASY and even Flexible backfilling which was designed to solve problems involving deadlines and machine usage. Also, the schedule-based solution can easily follow the incremental approach keeping the algorithm runtime low in contrast to the situations when the schedule is re-computed from scratch upon each job arrival. From this point of view, the non-incremental schedule-based methods are much more time consuming since their runtime is growing with the size of the schedule very quickly.

In the future, current model should be extended with a network simulation and also include a certain level of uncertainty such as the dynamic resource changes or the time estimations of the job execution, since the precise job execution times were available to every evaluated algorithm in the current simulations. Next, the effect of such uncertainty on the performance of the schedule-based methods will be studied. In Grids, the uncertainty and imprecision of information together with the dynamic changes represent a more realistic scenario. Usually, this is not a crucial issue for the simplest queue-based algorithms because they are designed to deal with dynamic changes and often require a very limited amount of information at the cost of limited performance. Our schedule-based approach, the backfilling (Tsafrir and Feitelson, 2006) and generally every technique that aims to guarantee certain behavior — for example inclusion of advanced reservation — relies on the precision of available information more. Without that, the reliability of the computed schedule is limited, thereby some action such as local change or limited rescheduling must be done to keep the schedule up to date when an unexpected event occurs.

6 Acknowledgments

This work was kindly supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research intent No. 0021622419 and by the Grant Agency of the Czech Republic with grant No. 201/07/0205.

References

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

David Abramson, Rajkumar Buyya, Manzur Murshed, and Srikumar Venugopal. Scheduling parameter sweep applications on global Grids: A deadline and budget constrained cost-time optimisation algorithm. *International Journal of Software: Practice and Experience (SPE)*, 35(5):491–512, 2005.

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

Ranieri Baraglia, Renato Ferrini, and Pierluigi Ritrovato. A static mapping heuristics to map parallel applications to heterogeneous computing systems: Research articles. *Concurrency and Computation: Practice and Experience*, 17(13):1579–1605, 2005.

Rajkumar Buyya and Manzur Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, 14:1175–1220, 2002.

Gabriele Capannini, Ranieri Baraglia, Diego Puppin, Laura Ricci, and Marco Pasquali. A job scheduling framework for large computing farms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10. ACM, 2007.

Moab workload manager administrator's guide, version 5.3.0. Cluster Resources, December 2008. http://www.clusterresources.com/products/mwm/docs/.

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

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

Fred W. Glover and Gary A. Kochenberger, editors. *Handbook of metaheuristics.* Kluwer, 2003.

Fred W. Glover and Manuel Laguna. *Tabu search.* Kluwer, 1998.

Holger H. Hoos and Thomas Stützle. *Stochastic Local Search Foundations and Applications.* Elsevier, 2005.

Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in HPC resource management systems: Queuing vs. planning. In *9th International Workshop, JSSPP 2003*, volume 2862 of *LNCS*, pages 1–20. Springer, 2003.

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

James Patton Jones. *PBS Professional 7, administrator guide.* Altair, April 2005.

Dalibor Klusáček, Luděk Matyska, and Hana Rudová. Alea – Grid scheduling simulation environment. In *7th International Conference on Parallel Processing and Applied Mathematics (PPAM 2007)*, volume 4967 of *LNCS*, pages 1029–1038. Springer, 2008a.

Dalibor Klusáček, Hana Rudová, Ranieri Baraglia, Marco Pasquali, and Gabriele Capannini. Comparison of multi-criteria scheduling techniques. In *Grid Computing Achievements and Prospects*, pages 173–184. Springer, 2008b.

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

Edi Shmueli and Dror G. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In *9th International Workshop, JSSPP 2003*, volume 2862 of *LNCS*, pages 228–251. Springer, 2003.

Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. The EASY - LoadLeveler API Project. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47. Springer, 1996.

Warren Smith, Ian Foster, and Valerie Taylor. Scheduling with advanced reservations. In *International Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 127–132, 2000.

Karl-Uwe Stucky, Wilfried Jakob, Alexander Quinte, and Wolfgang Süß. Solving scheduling problems in Grid resource management using an evolutionary algorithm. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *LNCS*, pages 1252–1262. Springer, 2006.

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

Wolfgang Süß, Wilfried Jakob, Alexander Quinte, and Karl-Uwe Stucky. GORBA: A global optimising resource broker embedded in a Grid resource management system. In *International Conference on Parallel and Distributed Computing Systems, PDCS 2005*, pages 19–24. IASTED/ACTA Press, 2005.

Xueyan Tang and Samuel T. Chanson. 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*, pages 373 – 382. IEEE, 2000.

Ariel David Techiouba, Gabriele Capannini, Ranieri Baraglia, Diego Puppin, and Marco Pasquali. Backfilling strategies for scheduling streams of jobs on computational farms. In *Making Grids Work*, pages 103–115. Springer, USA, 2008.

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

Dan Tsafrir and Dror G. Feitelson. The dynamics of backfilling: solving the mystery of why increased inaccuracy may help. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 131–141. IEEE, 2006.

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

Figure captions

- Figure 1: The influence of the TS iterations and the TS execution frequency on the algorithm runtime (left), the machine usage (middle) and the number of delayed jobs (right).
- Figure 2: Number of delayed jobs (left), the machine usage (middle) and the makespan (right).
- Figure 3: The average algorithm runtime per one job (left) and the average job slowdown (right).