Scheduling Mixed Workloads in Multi-grids: The Grid Execution Hierarchy

Mark Silberstein, Dan Geiger and Assaf Schuster
Technion, Israel
{marks,dang,assaf}@cs.technion.ac.il

Miron Livny
University of Wisconsin, Madison, USA
miron@cs.wisc.edu

Abstract

Consider a workload in which massively parallel tasks that require large resource pools are interleaved with short tasks that require fast response but consume fewer resources. We aim at achieving high throughput and short response time when scheduling such a workload over a set of uncoordinated grids of varying sizes and performance characteristics.

We propose the concept of a grid execution hierarchy, where available grids are sorted according to their size, and the execution overheads increase with the size of the grids. We devise a scheduling algorithm for this execution hierarchy of grids by adapting the multilevel feedback queue approach to a multi-grid environment. The algorithm finds a grid of the size, availability, and overhead that best matches a task's resource requirements and expected turnaround time. Our approach is inspired by the Shortest Processing Time First policy (SPTF), in the sense that the task's processing demands are constantly reevaluated during its run, so that a task is migrated to a more suitable level of the execution hierarchy when appropriate.

We evaluate our approach in the context of the Superlink-online *system for processing genetic linkage analysis tasks – a production system consisting of several grids and utilizing tens of thousands of CPU hours a month [32]. With our approach the system provides nearly interactive response time for shorter tasks, while simultaneously serving throughput-oriented massively parallel tasks in an efficient manner* [1].

1 Introduction

Successful utilization of large-scale environments for running computationally demanding tasks has motivated scientists to adapt their applications for execution on grid platforms. However, the vision of the grid as a virtual computer of unlimited capacity is yet to materialize. Rather, access is often granted to multiple uncoordinated resource pools (which we call *grids*) that vary significantly in their size and performance characteristics.

For example, researchers often have access to specialized computational clusters of a few dozen CPUs, in addition to having a few machines dedicated to their research. Organization-wide grids usually allow utilization of idle cycles of many desktop computers and offer a total of several thousand non-dedicated CPUs. National and international grids, which may include several supercomputing centers, typically scale up to tens of thousand CPUs [3]. Finally, SETI@HOME-like communities [6] can potentially harvest cycles from hundreds of thousands of CPUs.

We aim at integrating all the grids accessible to a researcher into a single system that will execute a stream of tasks having vastly different computational requirements. We consider divisible load tasks, which can be divided and sub-divided into any number of asynchronous sub-tasks, called *jobs*. The system receives a stream of such tasks, where the number of operations each task has to perform, called the *task complexity*, is unknown, although its distribution is strongly biased towards lower complexity (or, *shorter*) tasks. The tasks in the task stream may have vastly different complexities, imposing a *mixed workload* on the executing environment. To achieve reasonable turnaround time, e.g, a few minutes for the shorter tasks and a few days for those of higher complexity, the *appropriate parallelization level* for a task is dictated by the task's complexity.

An important factor in the performance of a multi-grid system is the choice of a grid for task execution. While the number of CPUs in a grid is critical for obtaining high performance for higher complexity massively-parallel tasks, the response time for shorter tasks depends strongly on the execution overheads of the chosen grid.

Such overheads may vary significantly in different grids. While smaller grids are usually used exclusively by a small group of researchers and employ dedicated resources, larger grids are typically shared by hundreds or thousands of users, providing limited quality of service guarantees and being more vulnerable to attacks and failures. In fact, in the common case, overhead and availability seem to trade with the

[1] This work is supported by the Israeli Ministry of Science and the Israel Science Foundation.

grid size. We identify five sources of overheads which commonly increase the cost of running a task on larger grids:

- Slow and unreliable WAN connections due to geographic and organizational dispersion.
- Complicated resource management due to a large number of resources to be managed. Sometimes (as in EGEE [3]) a task will pass through several resource brokers and queue managers until it is assigned physical resources.
- Enforcement of rigid policies due to a large number of users (hundreds or thousands), making it hard for an individual user to improve her priority and gain access to more resources.
- Extensive security mechanisms such as authorization, authentication, and data encryption.
- High volatility of resources due to frequent occasional failures and task evictions in favor of higher priority users.

In the common case, these considerations result in reduced overheads for scheduling, invocation, and execution on smaller grids, allowing for more predictable execution and faster response. Larger grids, however, are usually tuned to provide high throughput, sometimes at the expense of higher turnaround times and less responsiveness.

The problem of using multi-grid environments can be solved by unifying all available grids into a large, flat grid, managed by one of the popular meta-schedulers [1, 4]. This solution, which is common in large-scale grid environments such as EGEE [3], typically uses a first-come first-served (FCFS) policy for a given user: tasks are opportunistically scheduled for execution on available resources that may reside at several different grids. However, this solution may result in all available resources being occupied by an early-arrived demanding task, thus delaying the execution of late arrivers and degrading system response for short tasks.

In order to handle mixed workloads, many deployments use a natural extension of the flat approach: short tasks are prioritized by assignment to different FCFS queues [29]. If a high complexity task occupies all available resources, some of them will be relinquished in favor of short higher-priority tasks. However, this approach assumes a priori knowledge of task complexity.

Further extension of the flat approach, similar to the multilevel feedback queue (MQ) [26], does not require knowledge of task complexity. It schedules every task at the highest priority queue and moves it to a lower priority queue if the task fails to complete within the queue time limit. In this way, a task will end up being assigned the correct priority according to its complexity. However, high complexity tasks may still be assigned to low-overhead grids, leaving only high-overhead resources for short tasks, which may result in unacceptable turnaround times.

In this paper we propose to combine MQ with a new concept of *the grid execution hierarchy*. All available grids are sorted according to their size and overhead: upper levels of the hierarchy include smaller grids with faster response time, whereas lower levels consist of one or more large-scale grids with higher execution overhead. A mixed workload task stream is scheduled on the hierarchy, so that each task is executed at the level that matches the task's complexity. As the complexity increases, so do the computational requirements and the execution overhead that can be tolerated. Consequently, the task to be executed will be placed at a lower level of the hierarchy.

A task is first placed at the highest level of the hierarchy, as its complexity is not known upon arrival. If a task fails to complete within the time limit of that level, it is migrated to a lower level of the hierarchy where more resources are available. This process continues until the last hierarchy level is reached or the task terminates.

One may wonder as to the reason for searching for the execution level starting from the top of the hierarchy. Indeed, the proper execution level for a given task is easy to determine if the task complexity is simple to compute. However, for the important class of applications considered in this paper, even estimating task complexity is in itself a demanding computational problem. Applications in this class include constraint processing, Bayesian networks inference, and other NP-hard problems, where task complexity estimation is NP-hard [16]. For such applications there exist heuristic algorithms that yield an upper bound on the task complexity, whose precision improves the longer they execute [20]. In our grid execution hierarchy framework, task complexity is reassessed at each level prior to the execution. If the complexity is within the queue complexity range, the task is executed at that level; otherwise it is moved to a lower level. The lower the level, the greater the amount of resources allocated for estimating more precisely its complexity, resulting in a better matched grid size.

The characteristics of our approach are highlighted in Table 1, where results (averaged over several attempts) of running two tasks, submitted one after another, are given: a long task of approximately six hours, parallelized into 3000 jobs and executed on a large Condor pool, and a short task of approximately thirty seconds on a single CPU. The tasks are scheduled using either MQ or FCFS. The grids are organized either as a large pool containing all resources (flat), or as a hierarchy (H). Obviously, turnaround time for the longer task is approximately the same for all four combinations of scheduling algorithm and system organization. In contrast, the turnaround time for the shorter task is affected by the higher priority it is assigned by MQ, and is further

	FCFS	FCFS+H	MQ	MQ+H
Long task	6.2h	6.4h	6.1h	6.3h
Short task	4.7h	4.2h	3.5m	44s

Table 1. Results of runs in multiple scheduling scenarios

improved by hierarchical organization, which ensures its assignment to highly responsive resources.

We evaluate the grid-hierarchy scheduling algorithm in the context of *Superlink-online*, a production system which assists geneticists worldwide in analyzing experimental results through genetic linkage analysis [32]. Superlink-online utilizes about 2700 CPUs located in Condor pools at the Technion in Haifa, and at the University of Wisconsin in Madison. Tasks, submitted via an Internet portal, go through complexity estimation, parallelization, and scheduling on the grids comprising the system.

The analysis of the traces of this production system shows that the proposed grid-hierarchy scheduling algorithm is able to distinguish tasks of different complexities, and assign them to a grid of appropriate power and overhead. Consequently, even when the system is overloaded with tasks of high complexity, it is still able to support fast, almost interactive turnaround times for short tasks, and reasonable completion times for medium complexity tasks.

The rest of the paper is organized as follows. In the next section we describe the model of the execution environment and the expected workload. We then present the grid-hierarchy scheduling algorithm, followed by implementation details of the production system with which this algorithm is evaluated. Our evaluation is based on the statistics for about 2300 tasks submitted to the Superlink-online system between June and December 2005 by users worldwide. We conclude with an overview of the related research and a discussion of further research.

2 Model

In this section we describe the platform, application and submission models.

2.1 Platform model

Grids are managed by local workload and resource management software that cannot be changed or reconfigured in any way. Jobs are submitted for execution in a standard way, through a front-end submission node, and are subject to local policies of the grid. No communication is assumed between the grid resources and the outside world except for submission nodes, due to firewalls separating uncoordinated grids. Faults, crashes, and other related events are handled by the local resource management software.

2.2 Application model

Tasks can be divided and sub-divided into any number of independent asynchronous jobs (divisible load tasks [12]). Tasks are parallelized using the master-worker paradigm, where a single master dynamically schedules tasks to multiple workers (see, for example, [22]). Migration of master-worker tasks across grids is supported via checkpoint/restart operation of the master component alone. Namely, the master preserves the results of previously terminated jobs as well as the state of its work queue across invocations. Although complete support for checkpoint/restart is desirable, this partial functionality is usually sufficient and more practical.

2.3 Submission model

Tasks are submitted independently, forming an incoming stream. The task complexity distribution is heavily biased towards short tasks (e.g.,[18, 19, 29]). Statistics gathered from our Superlink-online [32] production system for 2300 tasks show a similar bias (see Figure 7).

Upon submission, task complexity is unknown. However, an upper bound on task complexity can be computed by a procedure whose accuracy improves together with the amount of resources allocated for its execution, as is the case in the context of genetic linkage analysis (e.g., [20]).

3 Grid-hierarchy scheduling algorithm

The algorithm has two complementary components: organization of multiple grids as a grid execution hierarchy and procedures for scheduling tasks on this hierarchy.

3.1 Grid execution hierarchy

The purpose of the execution hierarchy is to classify available grids according to their performance characteristics, so that resources at each level of the hierarchy provide the best performance for tasks of a specific complexity range. In other words, a task of any complexity has a level in the hierarchy which best matches its needs in terms of overhead and available resources.

The upper levels of the hierarchy include smaller grids with faster response time, whereas lower levels consist of one or more large-scale grids with higher execution overhead. The number of levels in the hierarchy depends on the expected distribution of task complexities in the incoming task stream, as explained in Section 5.

Each level of the execution hierarchy is associated with a set of one or more queues. Each queue is connected to one or more grids at the corresponding level of the hierarchy, allowing submission of jobs into these grids. A task arriving at a given hierarchy level is enqueued into one of the queues. It can be either executed on the grids connected to that queue (after being split into jobs for parallel execution), or migrated to another queue at the same level by employing simple load balancing techniques, described later. If a task does not match the current level of the execution hierarchy, as determined by the scheduling procedure presented in the next subsection, it is migrated to a queue at the next lower level of the hierarchy.

3.2 Scheduling tasks in a grid hierarchy

The goal of the scheduling algorithm is to find the proper execution hierarchy level for a task of a given complexity with minimum overhead. Ideally, if we knew the complexity of each task and the performance of each grid in the system, we could compute the execution time of a task on each grid, placing that task on the one that provides the shortest execution time. In practice, however, neither the task complexity nor the grid performance can be determined precisely. Thus, the algorithm attempts to schedule a task using approximate estimates of these parameters, dynamically adjusting the scheduling decisions if these estimates turn out to be incorrect.

We describe the algorithm in steps, starting with the simple version, which is then enhanced.

3.2.1 Simple MQ with grid execution hierarchy

Each queue in the system is assigned a maximum time that a task may stay in the queue (queueing time) T_q, and a maximum time that a task may execute in the queue (execution time) T_e, with $T_e \leq T_q$ [2]. The queue configured to serve the shortest tasks is connected to the highest level of the execution hierarchy, the queue for somewhat longer tasks is connected to the next level, and so on.

A task is assumed to be short and thus first submitted to the top level queue. Indeed, while nothing is known about its complexity upon submission, recall that the complexity distribution in the task stream is biased toward shorter tasks. If any of the queue limits is violated, a task is preempted and migrated to the next lower queue (the one submitting tasks to the next hierarchy level).

Such an algorithm ensures that any submitted task will eventually reach the hierarchy level that provides enough resources for longer tasks and fast response time for shorter tasks. In fact, this is the original MQ algorithm applied to a grid hierarchy.

[2]For simplicity we do not add the queue index to the notations of queue parameters, although they can be set differently for each queue.

3.2.2 Avoiding hierarchy level mismatch

The simplistic scheduling algorithm above fails to provide fast response to short tasks if a long task is submitted to the system prior to a short one. Recall that the original MQ is used in time-shared systems, and tasks within a queue are scheduled using preemptive round-robin, thus allowing fair sharing of the CPU time [26]. In our case, however, tasks within a queue are served in FCFS manner (though later tasks are allowed to execute if the task at the head of the queue does not occupy all available resources). Consequently, a long task executed in a queue for short tasks may make others wait until its own time limit is exhausted.

Quick evaluation of the expected waiting and running times of a task in a given queue can prevent the task from being executed at the wrong hierarchy level. This is accomplished as follows. Each queue is associated with a maximum allowed single task complexity C_e and a maximum queue workload complexity C_q, where queue workload complexity is defined as the sum of the complexities of the tasks in the queue. The queue complexity limits C_e and C_q are derived from the queue time limits T_e and T_q by optimistically assuming linear speedup, availability of all resources at all times, and resource performance equal to the average in the grid. The optimistic approach seems reasonable here, because executing a longer task in an upper level grid is preferred over moving a shorter task to a lower level grid, which could result in unacceptable overhead. The following naive relationship between a complexity limit C and a time limit T reflects these assumptions:

$$C \quad = \quad \phi(T) = T \cdot (N \cdot P \cdot \beta), \qquad (1)$$

where N is the maximum number of resources that can be allocated for a given task, P is the average resource performance, and β is the efficiency coefficient of the application on a single CPU, defined as the portion of CPU-bound operations in the overall execution. By Eq. 1, $C_q = \phi(T_q)$ and $C_e = \phi(T_e)$.

The procedure **ProcessNewTask** in Figure 1 detects the tasks which exceed the queue complexity range and triggers their migration to a lower queue.

The procedure comprises three steps. The first step is to detect the queue overload, namely, whether the arriving task violates T_q. We denote by Q the estimate of the current queue workload complexity, as computed by the procedure **EnforceQueueLimits**, which is described later. We denote by C the current estimate of the task's complexity (C can be unknown for new tasks). If the violation is detected, the algorithm triggers an *overload* migration policy, which migrates the incoming task to the next queue or rejects it without estimating its complexity.

```
1. Overload detection
   if ($Q > C_q$) then call MigrationPolicy(overload)
2. Task complexity estimation
   if ($C$ is unknown or $C > C_e$) then
   $C \leftarrow$ {run complexity estimation for up to $\alpha T_e$ }
3. Migration or invocation
   if ($C > C_e$) then call
   MigrationPolicy(complexity mismatch)
   else leave for invocation in the current queue
```

Figure 1. Procedure ProcessNewTask

In the second step the task complexity is estimated [3]. Recall that we assume the availability of a complexity estimation procedure which produces an upper bound on the task's complexity. The longer this procedure executes, the more accurate is the bound. Allocating a small portion $\alpha < 1$ of T_e for complexity estimation often allows quick detection of a task that does not fit in the queue. However, the upper bound on the task's complexity might be much larger than the actual value. Consequently, if the task is migrated directly to the level in the hierarchy that serves the complexity range in which this estimate belongs, it may be moved to too low a level, thus decreasing the turnaround time. Therefore, the task is moved to the next level, where the complexity estimation algorithm is given more resources and time to execute, yielding a more precise estimate.

The final step ensures that tasks of complexities that are above C_e and would consequently fail to terminate within T_e, are not invoked in the queue. The *complexity mismatch* migration policy is triggerred for these tasks, migrating them to the next lower queue, unless the queue is at the lowest hierarchy level, in which case the task is rejected.

3.2.3 Enforcing queue limits

Using the procedure **ProcessNewTask** is insufficient to ensure that queue limits are not violated. The main reason for a task to stay in the queue longer than initially predicted is that the grid performance estimates may turn out to be too optimistic, as they do not account for the possible fluctuations of the number of resources due to failures, changes in grid load, and other factors. Thus, the queue time limits must be enforced in a manner similar to that used in the original MQ algorithm, by monitoring the queue and migrating tasks which violate the queue limits. Furthermore, because the complexity of remaining computations for a given task can be determined, tasks which are very likely to violate the queue limits in the future can be detected. Early detection of such tasks increases the chance that later tasks will complete without migration to lower levels.

[3]The complexity estimation can be quite computationally demanding for larger tasks, and in which case it is executed using grid resources.

```
For every task $j$, starting from the head of the queue
  1. Detect actual violation of queue limits
     if ( $t_q^j > T_q$ or $t_e^j > T_e$) then call
     MigrationPolicy(overdue)
  2. Detect future violation of queue limits
     (a) Obtain complexity of total remaining computations $C$
     (b) Check if enough time remains to complete the task
         if ($C > \phi(T_e - t_e^j)$ or $C > \phi(T_q - t_q^j)$)
         then call MigrationPolicy(task potential overdue)
     (c) Check if enough time remains with the preceding tasks
         if ($Q + C > C_q$) then call
         MigrationPolicy(queue potential overdue)
         else $Q = Q + C$
return $Q$
```

Figure 2. Procedure EnforceQueueLimits

This is accomplished by the **EnforceQueueLimits** procedure (Figure 2). Each task j stores its queuing time t_q^j (the time from arrival to the queue until termination) and its execution time t_e^j (the time from the moment the first job is started by the grid middleware until termination).

The procedure **EnforceQueueLimits** consists of three steps. The first step determines the tasks which actually violate the queue limits, triggering the *overdue* migration policy, which preempts and migrates these tasks to the next lower queue or terminates them if this is the lowest level. If a task is the only task in the queue, its migration (termination) is delayed until another task arrives.

The second step detects tasks which are likely to violate the queue limits. While such tasks have not exhausted their queue limits yet, they will likely be preempted in the future due to too many remaining computations (*2.b*).

The third step (*2.c*) prevents accumulation of tasks in the queue if the tasks at the head of the queue are progressing too slowly and causing later tasks to exhaust their queue limit.

Note that the procedure intentionally distinguishes the *overdue* and *task (queue) potential overdue* migration policies (*2.b*, *2.c*), to support different system behavior in each case. In the evaluation section we show how disabling these policies affects the queue performance.

3.3 Handling multiple grids at the same level of the execution hierarchy

The problem of scheduling in a configuration where multiple grids are placed at the same level of the execution hierarchy is equivalent to the well-studied problem of load sharing in multi-grids. It can be solved in many ways, including using the available meta-schedulers, such as [1], or flocking [33]. If no existing load sharing technologies can be deployed between the grids, we implement load sharing as follows.

Our implementation is based on a push migration mechanism (such as in [18]) between queues, where each queue is connected to a separate grid. Each queue periodically samples the availability of resources in all grids at its level of the execution hierarchy. This information, combined with the data on the total workload complexity in each queue, allows the expected completion time of tasks to be estimated. If the current queue is considered suboptimal, the task is migrated. Conflicts are resolved by reassessing the migration decisions at the moment the task is moved to the target queue. Several common heuristics are implemented to reduce sporadic migrations that may occur as a result of frequent fluctuations in grid resource availability [35]. Such heuristics include, among others, averaging of momentary resource availability data with the historical data, preventing migration of tasks with a small number of pending execution requests, and others.

4 The application

We have implemented the grid-hierarchy scheduling algorithm in an online distributed system for execution of genetic linkage analysis tasks, called *Superlink-online* [32]. The tasks, submitted via the Internet by geneticists from national and international medical research institutions, are scheduled and parallelized by the system for execution in the distributed environment.

4.1 Genetic linkage analysis

Genetic linkage analysis is a statistical tool used by geneticists to facilitate the process of identification of disease-provoking genes [30]. It is a well-established technique employed by geneticists in everyday practice. Successful identification of the affected genes helps provide better prevention and treatment for a disease, and reveals the functional characteristics of genes.

Performing genetic linkage analysis can be represented as the problem of computing an expression of the form

$$\sum_{x_1}\sum_{x_2}\cdots\sum_{x_n}\prod_{i=1}^{m}\Phi_i(\mathbf{X_i}), \qquad (2)$$

where $\mathbf{X} = \{x_1, x_2, \ldots, x_n | x_i \in \mathbb{N}\}$ is a set of non-negative discrete variables, $\Phi_i(\mathbf{X_i})$ is a function $\mathbb{N}^k \rightarrow \mathbb{R}$ from the subset $\mathbf{X_i} \subset \mathbf{X}$ of these variables of size k to the reals, and m is the total number of functions to be multiplied. Functions are specified by a user as an input. For more details we refer the reader to [21, 32].

4.2 Serial algorithm

The problem of computing Eq. 2 is known to be NP-complete [13]. One possible algorithm for computing this expression is called variable elimination [15].

The complexity of the algorithm is fully determined by the order in which variables are eliminated. Finding an optimal elimination order is NP-complete [7]. A close-to-optimal order can be found using the stochastic greedy algorithm proposed in [20]. The algorithm can be stopped at any point, and it produces better results the longer it executes, converging faster for smaller problems. The algorithm yields a possible elimination order and an upper bound on the problem complexity, i.e., the number of operations required to carry out the computations if that order is used. It is this feature of the algorithm that is used during the scheduling phase to quickly estimate the complexity of a given problem prior to execution.

4.3 Parallel algorithm and its execution

The algorithm for finding an elimination order consists of a large number of independent iterations, and is trivially parallelizable by distributing them over different CPUs using the master-worker paradigm.

Parallelization of the variable elimination algorithm also fits the master-worker paradigm and is performed as follows. We represent the first summation over x_1 in Eq.2 as a sum of the results of the remaining computations, performed for every value of x_1. This effectively splits the problem into a set of independent subproblems having exactly the same form as the original one, but with the complexity reduced by a factor approximately equal to the number of values of x_1. We use this principle recursively to create subproblems of a desired complexity. Each subproblem is then executed independently, with the final result computed as the sum of all partial results.

Parallel tasks are executed in a distributed environment via Condor [33], which is a general purpose distributed batch system, capable of utilizing idle cycles of thousands of CPUs. Condor hides most of the complexities of job invocation in an opportunistic environment. In particular, it handles job failures that occur because of changes in the system state. Such changes include resource failures, or a situation in which control of a resource needs to revert to its owner. Condor also allows resources to be selected according to the user requirements via a matching mechanism.

There are three stages in running master-worker applications in Condor: the parallelization of a task into a set of independent jobs, their parallel execution via Condor, and the generation of final results upon their completion. In our implementation, this flow is managed by the Condor flow execution engine, called DAGman, which invokes jobs according to the execution dependencies between them, specified as a directed acyclic graph (DAG). The complete genetic linkage analysis task comprises two master-worker applications, namely, parallel ordering estimation and parallel variable elimination. To integrate these two applications into a single execution flow, we use an outer DAG composed of two internal DAGs, one for each parallel application.

DAGman is capable of saving a snapshot of the flow state, and then restarting execution from this snapshot at a later time. We use this feature for migration of a task to another queue as follows: the snapshot functionality is triggered, all currently executing subtasks are preempted, the intermediate results are packed, and the task is transferred to another queue where it is restarted.

5 Deployment of Superlink-online

The current deployment of the Superlink-online portal is presented in Figure 3.

We configure three levels of the execution hierarchy, for the following reasons. About 60% of the tasks take a few minutes or less, and about 28% take less than three hours, as reflected by the histogram in Figure 7. This suggests that two separate levels, Level 1 and Level 2, should be allocated for these dominating classes, leaving Level 3 for the remaining longer tasks. Yet, the current system is designed to be configured with any number of levels to accommodate more grids as they become available.

Each queue resides on a separate machine, connected to one or several grids. Utilization of multiple grids via the same submission machine is enabled by the Condor flocking mechanism, which automatically forwards job execution requests to the next grid in the list of available (*flocked*) grids, if these jobs remain idle after previous resource allocation attempts in the preceding grids.

Queue Q1 is connected to the dedicated dual CPU server and invokes tasks directly without parallelization. Queue Q2 resides on a submission machine connected to the flock of two Condor pools at the Technion. However, due to the small number of resources at Q2, we increased the throughput at Level 2 of the execution hierarchy by activating load sharing between Q2 and Q3, which is connected to the flock of three Condor pools at the University of Wisconsin in Madison. The tasks arrive to Q3 only from Q2. Queue Q4 is also connected to the same Condor pools in Madison as Q3, and may receive tasks from both Q2 and Q3.

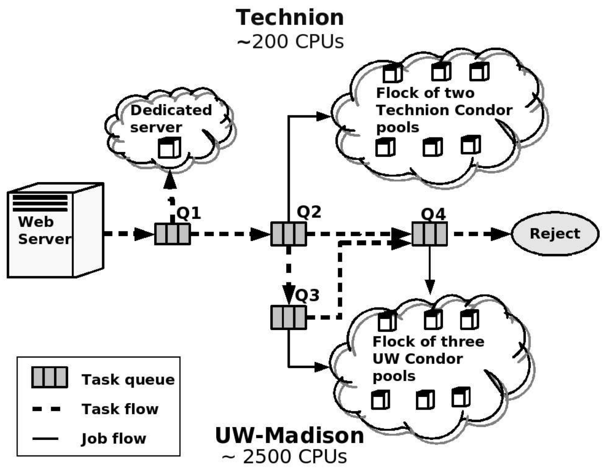

Figure 3. Superlink-online deployment

In fact, Q3 exhibits rather high invocation latencies (as can be observed from the overhead analysis in Figure 6), and does not fit Level 2 of the execution hierarchy well. Alternatively, Q3 could have been set as an additional level between Q2 and Q4, and the queue time limit of Q2 could have been adjusted to handle smaller tasks. However, because both queues can execute larger tasks efficiently, such partitioning would have resulted in unnecessary fragmentation of resources. Migration allows for a more flexible setup, which takes into account load, resource availability and overheads in both queues, and moves a whole task from Q2 to Q3 only if this yields better task performance. Typically, small tasks are not migrated. However, larger tasks are migrated, as usually benefit from execution in a larger grid as they are allocated more execution resources (see Table 3). This configuration results in better performance for smaller tasks than does flocking between these two grids, as it ensures their execution on the low-overhead grid.

To ensure that larger tasks of Q4 do not delay smaller tasks of Q3, jobs of tasks in Q3 are assigned higher priority and may preempt jobs from Q4. Starvation is avoided via internal Condor dynamic priority mechanisms [2].

The queue constraints T_q and T_e are configured as detailed in Table 2. The intuition behind the current configuration is as follows. The average time for allocation of a single CPU in the grid attached to Q2 is about 20 seconds. Thus, tasks arriving to this queue should be about ten times longer in order for the overhead not to dominate their performance. Consequently, the tasks below 200 seconds should be served in the previous queue, resulting in $T_e = 3$ minutes for Q1. Values of T_q are set to prevent tasks from being accumulated in queues, as the users prefer the tasks

Queue	T_q (min)	T_e (min)	N	P (KFlops)
Q1	6	3	2	817477
Q2	320	160	100	775251
Q3	160	160	500	649505
Q4	2880	2880	500	649505

Table 2. Parameters: T_q– queue waiting time limit, T_e– queue execution time limit, N– maximum available CPUs for a single user, P— average performance of computers in grid

to be rejected rather than delayed. We restrict the allowed queue length to up to two tasks of maximum duration for Q1, up to two tasks in Q2, and only one task of Q3 and Q4.

The maximum number of available CPUs for a single user is smaller than the total number of resources in the corresponding grids. Out of 200 CPUs in the Technion Condor pools, only 100 satisfy the minimum memory requirements of the application. For the Madison Condor pool, the limit of 500 jobs is due to the recommended value of the maximum number of running jobs concurrently handled by a single submission machine. More jobs cause severe overload of the submission machine and thus are avoided.

6 Results

We analyzed the traces of 2300 tasks, submitted to the Superlink-online system via Internet by users worldwide for the period between the 1st of June and the 31st of December 2005. During this time, the system utilized about 460,000 CPU hours (52.5 CPU years) over all Condor pools connected to it (according to the Condor accounting statistics). This time reflects the time that would have been spent if all tasks were executed on single CPU. About 70% of the time was utilized by 1971 successfully completed tasks. Another 3% was wasted because of system failures and user-initiated task removals. The remaining 27% of the time was spent executing tasks which failed to complete within the queue time limit of Q4, and were forcefully terminated. However, this time should not be considered as lost since users were able to use partial results. Still, for clarity, we do not include these tasks in the analysis.

6.1 Utilization of the execution hierarchy

We compared the total CPU time required to compute tasks by each level of the execution hierarchy relative to the total system CPU consumption by all levels together. As expected, the system spent most of its time handling the tasks at Level 3, comprising 82% of the total running time

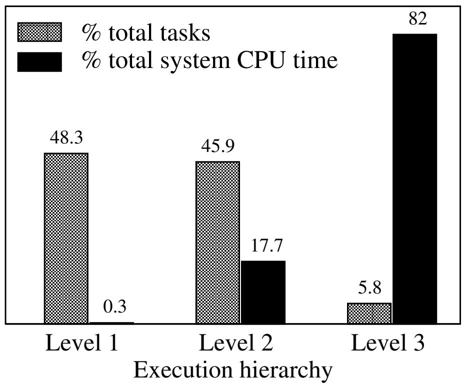

Figure 4. Portion of tasks handled by each level of the hierarchy (first column) versus portion of the overall system CPU time utilized by each level (second column)

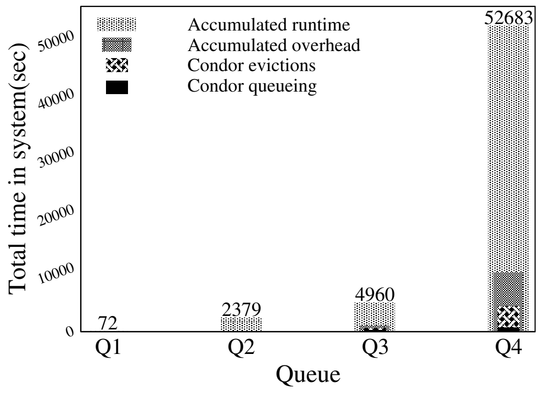

Figure 5. Average accumulated time (from arrival to termination) of tasks in each queue

of the system (see Figure 4). The tasks at Level 2 consumed only 17.7% of the total system bandwidth, and only 0.3% of the time was consumed by the tasks at Level 1. If we consider the total number of tasks served by each level, the picture is reversed: the first two levels served significantly more tasks than the lower level. This result proves that the system was able to provide short response time to the tasks which were served at the upper levels of the hierarchy.

This conclusion is further supported by the graph in Figure 5, which depicts the average accumulated time of tasks in each queue, computed from the time a task is submitted to the system until it terminates. This time includes accumulated overheads, which are computed by excluding the time of actual computations from the total time. As previously, the graph shows only the tasks which completed successfully. Observe that very short tasks which require less than three minutes of CPU time and are served by Q1 stay in the system only 72 seconds on average regardless of the load in the other queues.This is an important property of the grid-hierarchy scheduling algorithm.

The graph also shows average accumulated overhead for tasks in each queue, which is the time a task spent on any activity other than the actual computations.

The form of the graph requires explanation. Assuming a uniform distribution of task runtimes and availability of an appropriate grid for each task, the task accumulated time is expected to increase linearly towards lower levels of the hierarchy. However in practice these assumptions do not hold. There are exponentially more shorter tasks requiring up to 3 minutes on single CPU (see Figure 7). This induces a high load on Q1, forcing short tasks to migrate to Q2 and thus reducing the average accumulated time of tasks in Q2. This time is further reduced by the load sharing between Q2 and Q3, which causes larger tasks to migrate from Q2 to Q3. Thus, shorter tasks are served by Q2, while longer ones are executed in Q3, resulting in the observed difference between the accumulated times in these queues. To explain the observed steep increase in the accumulated time in Q4, we examined the distribution of running times in this queue. We found that shorter tasks (while exceeding Q3's allowed task complexity limit) were delayed by longer tasks that preceded them. Indeed, over 70% of the overhead in that queue is due to the time the tasks were delayed because of other tasks executing in that queue. This delay is a result of disabling the *potential overdue* migration policy in Q4, which is enabled in all other queues. Tasks in Q4 are allowed to run until they actually violate the queue time limits in order to allow generation of partial results, which are valuable in genetic linkage analysis applications. Thus, the queuing times of shorter tasks arriving to Q4 increase, resulting in longer tasks dominating the accumulated time. Availability of additional grids for the execution of higher complexity tasks would allow for the queueing and turnaround time to be reduced.

6.2 Overhead distribution in queues

Figure 6 provides a more detailed view of the types of overhead in each queue. This includes the invocation and control overheads incurred by DAGman as well as the time spent on complexity estimation, migration, and waiting for Condor to allocate resources. This last parameter is computed as the time from which the first job is submitted to the time when Condor starts executing it.

The major overhead of the tasks in Q1 is due to complexity estimation (11 seconds). In the initial implementation, a task was executed without complexity estimation, but preempted if it turned out to be a long task. However, since

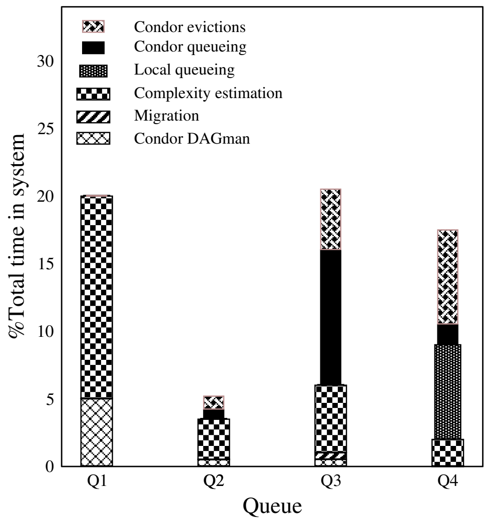

Figure 6. Overhead distribution in different queues

the tasks in our system are often submitted in bursts, this resulted in higher load on Q1 and delays for short tasks.

The task flow is managed via DAGman, the implementation of which carries its own 4 seconds, because DAGman sometimes requires few seconds to detect termination of jobs in the DAG. Note that Q1 invokes tasks locally, and thus does not suffer any Condor-related overheads.

Queues Q2 and Q3 serve tasks of about the same size and thus should exhibit comparable overheads. Indeed, the overheads due to DAGman and complexity estimation are almost equal, with the latter being slightly less in Q2 due to the availability of faster CPUs (see Table 2). However, Q3 shows significantly higher overheads than Q2 due to long Condor queueing times, reaching several minutes on average, versus 20 seconds on average for Q2.

Q4 shows the least overheads (3.5%) in terms of complexity estimation and waiting time in Condor queues, relative to the average task execution time in that queue. However, as was previously explained, insufficient resources caused some of the tasks to be delayed by the presence of long-running tasks that preceded them in the queue. This resulted in long delays due to local queueing.

An important factor in the overheads of Q3 and Q4 is the volatility of grid resources, namely, the loss of computations due to evictions. We note that this overhead is significantly higher for these queues, connected to the Condor pool at UW Madison, than for Q2, in which tasks are submitted to the Technion's small grid.

The graph in Figure 6 shows that in all queues, the overhead of the grid-hierarchy scheduling algorithm and its implementation does not exceed 20% of the total task time in

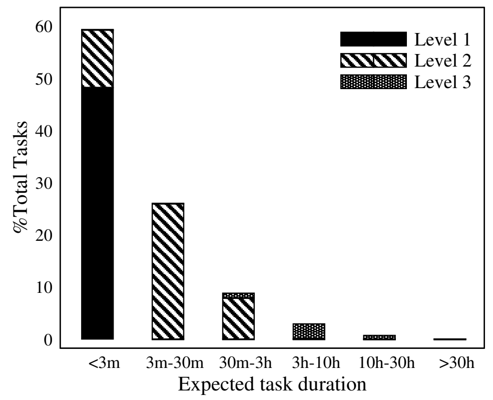

Figure 7. Actual distribution of tasks among different levels of hierarchy versus their expected duration. Ideally, each column would have a single level color.

the system even for very short tasks, and significantly less for longer ones, which is a reasonable trade-off for obtaining short response times.

6.3 Distribution of tasks in levels

Optimally, a task should be directly invoked at the best level in the hierarchy, acquiring the maximum possible resources at that level (which exactly matches our optimistic assumption in Eq. 1). Thus, we call *expected task duration* the time it would have taken the task to complete if it was immediately given the resources of the best matching level of the hierarchy upon entering the system. Figure 7 shows the distribution of the expected task durations, derived from their actual exact complexity, and the execution level in the hierarchy where these tasks were executed in the real system. Ultimately, each class of tasks is supposed to be handled by a level of the hierarchy matching its real complexity. However, about 11% of the tasks that are expected to be processed by Level 1 of the hierarchy are migrated and executed at Level 2. Further investigation revealed two reasons: 1) high load in the queue Q1 of Level 1, resulting in automatic offloading of tasks to Q2; 2) too loose an upper bound on the complexity produced at the first stage. As opposed to Level 1, Level 2 performs well for all jobs with expected duration below its time limit, with only few longer tasks moved to Level 3. We found that these tasks were initially invoked at Level 2, but were later moved due to momentary Condor failures. These failures resulted in low availability of resources and caused the tasks to be preempted in accordance with the *potential overdue* migration policy.

Average	**Q2**	**Q3**	**Q4**
Volatility (%of runtime)	1	4.5	7
Volatility (%of submitted tasks)	1	7	14
Number of allocated resources per task (absolute)	55	81.7	141
Number of allocated resources per task (% of requests)	50	54	9
Number of jobs per task	93	194	1600
Job duration (min)	9.4	13.3	15.3

Table 3. Resource and task properties in different queues

6.4 Level of parallelism and volatility

Running parallel tasks in a grid environment is complicated by the inherent volatility of the resources. A job can be evicted at any point of execution and then restarted on another resource, either from the beginning or from the last checkpoint, if such functionality is supported. Jobs in our application do not support checkpointing, and thus were shortened to minimize the overhead due to evictions.

Table 3 shows the effect of resource volatility on the performance of our tasks in each queue. We considered only parallel tasks, namely, those which consisted of multiple jobs, and the values are averaged over all tasks in a given queue. We found that for a task running in the Technion Condor pool, about 1% of the task's running time is lost due to evictions (row 1), and 1% of its jobs are evicted (row 2). For similar tasks in Q3 these values are 4.5% and 7% respectively. Since the total average runtime of tasks as well as job durations are similar in both queues, resources in the Madison Condor pool seem to exhibit a higher degree of volatility, confirming our assumption of the size-volatility trade-off. This is further confirmed by tasks in Q4, where about 14% of all jobs of the task are usually evicted. This value reflects an average of the fluctuations of resource volatility over longer periods of execution of these tasks.

Simple calculations show that, provided a dedicated cluster, a task at Q2 could have been completed in about 20 minutes, while the actual results in the opportunistic environment are about 4 times higher. This is in part due to the known phenomenon where, when short tasks are executed in opportunistic environments, the last few jobs of a task may dominate its running time because of resource volatility and heterogeneity [27].

Another important grid property is the amount of resources simultaneously allocated to the same user (row 3 in Table 3). We measured this value for each parallel task by counting the number of simultaneously executing jobs between invocations of the first and the last submitted jobs, sampling at any invocation or termination event and averaging over all available samples. The intuition is to measure the resource allocations only when a task still has pending job execution requests. We note that this value depends on the total number of job execution requests of a given task. Thus we normalize it by the total number of jobs per task, and average it over all parallel tasks in the queue (row 4 in Table 3). These values show that on average, tasks which are scheduled for execution at UW Madison obtain more resources than those scheduled in the Technion Condor pools, justifying the structure of our grid hierarchy.

7 Related work

Execution of parallel tasks in grid environments has been thoroughly studied by grid researchers.

Running massively parallel tasks in heterogeneous large-scale environments has been subject of many works (e.g., [10, 36, 11, 24, 9, 27]), that strive to minimize the turnaround time of a single task. In particular, [27] addressed the problem of resource management for short-lived tasks on desktop grids, demonstrating the suitability of grid platforms for execution of short-lived applications.

Meta-schedulers, such as [1, 4], strive to maximize the overall throughput and system utilization, as opposed to minimizing the task's turnaround time in our work.

Sabin et al. [31] suggest an algorithm for scheduling a stream of parallel applications in a multi-grid, assuming availability of reservation capabilities and absence of resource failures. Marchal et al. [28] discuss steady-state scheduling of divisible workloads in wide-area grids. However, according to the authors, steady-state analysis ignores the initialization and cleanup phases, which are critical for short-lived tasks. The meta-scheduler component of the GrADS project [14, 34] supports scheduling of multiple tasks. This work assumes the ability to directly invoke and preempt jobs on a given resource, and is not applicable to our case. Still, it highlights the importance of scheduling of multiple tasks for improving the turnaround time of individual tasks in grids. Another component of this project is the rescheduler [35], which inspired our implementation of load sharing between queues.

The work in [18] demonstrates the benefits of load sharing in a grid comprised of independently managed supercomputers, where entire parallel tasks are migrated between sites upon decisions made distributively by each queue. This work also influenced our load sharing mechanisms between the same-level grids.

A multilevel feedback queue algorithm for time-shared single-CPU systems appeared first in [26]. The authors of [23, 17] analyze the scheduling of tasks with highly variable known processing times on a set of identical servers. The authors show both theoretically and via simulation that a scheduling policy which minimizes the waiting time in the system is one in which each server is assigned tasks of a specific size range, approximating SPTF scheduling. This principle is applied in many production supercomputing environments (e.g., [25, 19, 29]). Results published in [8] show that SPTF policies do not penalize long tasks, when the task size distribution has a heavy-tail property and the largest 1% of the tasks comprises more than half the load, as in our system. Although these works assume availability of task size information and homogeneous servers, they encouraged us to apply similar techniques in our system.

8 Conclusions and future work

In this work we presented a method for organizing grids and an algorithm for scheduling mixed workloads in multi-grid environments. We implemented the algorithm for the Superlink-online production system [5, 32] and demonstrated that it yields short response times for short tasks even when the system is already loaded with long ones.

Future work will address several limitations of the current scheme. While the static approach we now use for building the execution hierarchy yields reasonable performance, the volatile nature and properties of grid systems call for dynamic structures. This requires on-the-fly adaptation of the hierarchy to the changing properties of the grids, and a cost model to take into account locality of applications and execution platforms. More precise predictions and/or estimations of application runtime and task complexity are needed. These should take into account real-time status information about the grids.

9 Acknowledgments

We thank Anna Tzemach and Nikolay Dovgolevsky for implementation of the Web interface and the initial parallelization of the genetic linkage software, the Condor team in Madison for their support, Dr. Alejandro Shäffer for user feedback, and Prof. Dick Epema for valuable remarks.

References

[1] Community scheduler framework. http://www.globus.org/toolkit/docs/4.0/contributions/csf.

[2] Condor manual. http://www.cs.wisc.edu/condor/manual/v6.7.

[3] Enabling grids for e-science. http://public.eu-egee.org.

[4] Moab grid suite. http://www.clusterresources.com/pages/products/moab-grid-suite.php.

[5] Superlink-online genetic linkage analysis system. http://bioinfo.cs.technion.ac.il/superlink-online.

[6] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *GRID 2004*, pages 4–10, 2004.

[7] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Algorithms and Discrete Methods*, 8:277–284, 1987.

[8] N. Bansal and M. Harchol-Balter. Analysis of srpt scheduling: investigating unfairness. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'01)*, pages 279–290, New York, NY, USA, 2001. ACM Press.

[9] O. Beaumont, A. Legrand, and Y. Robert. Scheduling divisible workloads on heterogeneous platforms. *Parallel Computing*, 29(9):1121–1152, 2003.

[10] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, pages 100–111, Washington, DC, USA, 1996. IEEE Computer Society.

[11] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[12] V. Bharadwaj, T. G. Robertazzi, and D. Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[13] G. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

[14] H. Dail, O. Sievert, F. Berman, H. Casanova, A. YarKhan, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. Scheduling in the grid application development software project. *Grid Resource Management: State-of-the-art and Future Trends*, pages 73–98, 2004.

[15] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In J. M. I. (Ed.), editor, *Learning in Graphical Models*, pages 75–104. Kluwer Academic Press., 1998.

[16] R. Dechter. *Constraint Processing.* Morgan Kaufmann, San Francisco, 2003.

[17] D. G. Down and R. Wu. Multi-layered round robin routing for parallel servers. Technical Report 2004-023, EURANDOM Technical Report, 2004.

[18] D. England and J. Weissman. Costs and benefits of load sharing in the computational grid. In D. G. Feitelson and L. Rudolph, editors, *10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[19] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997.

[20] M. Fishelson, N. Dovgolevsky, and D. Geiger. Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, 59:41–60, 2005.

[21] N. Friedman, D. Geiger, and N. Lotner. Likelihood computation with value abstraction. In *16th Conference on Uncertainty in Artificial Intelligence (UAI'00)*, pages 192–200. Morgan Kaufmann, 2000.

[22] J. Goux, J. Linderoth, and M. Yoder. Metacomputing and the master-worker paradigm, 1999.

[23] M. Harchol-Balter, M. E. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel Distributed Computing*, 59(2):204–228, 1999.

[24] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In *GRID 2000*, pages 214–227, 2000.

[25] S. Hotovy, D. Schneider, and T. O'Donnell. Analysis of the early workload on the Cornell Theory Center's IBM SP2. Technical Report 96TR234, Cornell Theory Center, 1996.

[26] L. Kleinrock and R. Muntz. Processor sharing queueing models of mixed scheduling disciplines for time shared systems. *Journal of ACM*, 19:464–482, Jul 1972.

[27] D. Kondo, A. A. Chien, and H. Casanova. Resource management for rapid application turnaround on enterprise desktop grids. In *ACM/IEEE Conference on Supercomputing (SC'04)*, page 17, Washington, DC, USA, 2004. IEEE Computer Society.

[28] L. Marchal, Y. Yang, H. Casanova, and Y. Robert. Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms. *International Journal of High Performance Computing Applications*, 2006, to appear.

[29] E. Medernach. Workload analysis of a cluster in a grid environment. In D. G. Feitelson, E. Frachtenberg, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 36–61. Springer Verlag, 2005.

[30] J. Ott. *Analysis of Human Genetic Linkage.* Johns Hopkins University Press, Baltimore, Maryland, 1999.

[31] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environment. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 87–104. Springer Verlag, 2003.

[32] M. Silberstein, A. Tzemach, N. Dovgolevsky, M. Fishelson, A. Schuster, and D. Geiger. On-line system for faster linkage analysis via parallel execution on thousands of personal computers. *Americal Journal of Human Genetics*, 2006, in press.

[33] D. Thain and M. Livny. Building reliable clients and servers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San-Francisco, 2003.

[34] S. Vadhiyar and J. Dongarra. A metascheduler for the grid. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

[35] S. Vadhiyar and J. Dongarra. Self adaptivity in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, 2005.

[36] Y. Yang and H. Casanova. Rumr: Robust scheduling for divisible workloads. In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 114, Washington, DC, USA, 2003. IEEE Computer Society.