Heuristics for Scheduling Parameter Sweep Applications in Grid Environments

Henri Casanova* Arnaud Legrand† Dmitrii Zagorodnov* Francine Berman*

* Computer Science and Engineering Department
University of California, San Diego, USA
[casanova,dzagorod,berman]@cs.ucsd.edu

† Laboratoire de l'Informatique et du Parallélisme
École Normale Supérieure de Lyon, France
alegrand@ens-lyon.fr

Abstract

The Computational Grid provides a promising platform for the efficient execution of parameter sweep applications *over very large parameter spaces. Scheduling such applications is challenging because target resources are heterogeneous, because their load and availability varies dynamically, and because independent tasks may share common data files. In this paper, we propose an adaptive scheduling algorithm for parameter sweep applications on the Grid. We modify standard heuristics for task/host assignment in perfectly predictable environments (*Max-min, Min-min, Sufferage*), and we propose an extension of Sufferage called* XSufferage. *Using simulation, we demonstrate that XSufferage can take advantage of file sharing to achieve better performance than the other heuristics. We also study the impact of inaccurate performance prediction on scheduling. Our study shows that: (i) different heuristics behave differently when predictions are inaccurate; (ii) increased adaptivity leads to better performance.*

1. Introduction

Fast networks make it possible to aggregate CPU, network and storage resources into *Computational Grids* [8]. Such environments can be used effectively to support very large-scale runs of distributed applications. An ideal class of applications for the Grid is the class of *parameter sweep applications*, applications structured as a set of multiple "experiments", each of which is executed with a distinct set of parameters.

Executing a parameter sweep on the Grid involves the assignment of tasks to resources. Although the experiments (or *tasks*) of a parameter sweep application are independent, a number of issues make scheduling such applications challenging. First, resources in the Grid are typically *shared* so that the contention created by multiple applications creates dynamically fluctuating delays and qualities of service. In addition, Grid resources are *heterogeneous* and may not perform similarly for the same application. Moreover, although parameter sweep tasks are independent, they may share common input files which reside at remote locations, hence the performance-efficient assignment and scheduling of the application must include consideration of the impact of data transfer times. Previous work [3] has demonstrated that run-time, adaptive, application-scheduling based on dynamic information about the status of computing resources is a good general approach for achieving performance on the Grid.

In [20], three heuristics (*Max-min, Min-min* and *Sufferage*) were proposed for the scheduling of independent tasks in single-user, homogeneous environments. **In this paper, we modify existing heuristics to schedule parameter sweep applications with file I/O requirements, we propose an extended version of Sufferage, XSufferage, and we study the impact of inaccurate performance prediction on scheduling**. We integrate these heuristics into a gen-

This research was supported in part by NSF Grant ASC-9701333, NASA/NPACI Contract AD-435-5790, DARPA/ITO under contract #N66001-97-C-8531, and CNRS/INRIA project ReMaP

eral adaptive scheduling algorithm and compare them in various simulated computing environments and for various application scenarios. We will use a standard performance metric to evaluate our heuristics: the application *makespan* [22], i.e. the time between the first input files is sent to a computational server and the last output file is returned to the user. Our ultimate goal is to include our adaptive scheduling algorithm in a software framework, a Parameter Sweep Template (PST), developed as part of the AppLeS project [13]. PST will be the subject a a future paper.

In a Grid environment it is usually difficult to obtain accurate predictions for computing and networking resource performance; moreover most scheduling heuristics make use of such predictions. We designed a simulator that allows us to experiment with different levels of performance prediction accuracy. In this paper we present a preliminary study of the effect of increasing inaccuracy on the heuristics under consideration and discuss how adaptivity can be used to promote performance in Grid environments.

This paper is organized as follows. In Section 2, we present our models for both the application and the underlying Grid environment. In Section 3, we present our scheduling algorithm. Section 4 focuses on the different task/host assignment heuristics whereas Sections 5 discusses adaptivity and performance prediction accuracy. Section 7 references related research work, and Section 8 concludes the paper.

2. A Scheduling Model for Parameter Sweeps on the Grid

2.1. Application Model

We define a **parameter sweep** application as a set of n independent sequential tasks $\{T_i\}_{i=1,..,n}$. By *independent* we mean that there are no inter-task communications or data dependencies (i.e. task precedences). We assume that the input to each task is a set of files and that a single file might be input to more than one task. In our model, without loss of generality, each task produces exactly one output file. Figure 1 shows an example with input file sharing among tasks. We assume that the size of each input and output file is known a-priori.

This model is motivated by our primary target application for PST: MCell [29], a micro-physiology application that uses 3-D Monte-Carlo simulation techniques to study molecular bio-chemical interactions within living cells. An MCell run is composed of multiple Monte-Carlo simulations for cell regions whose geometries are described in (potentially very large) files. For instance,

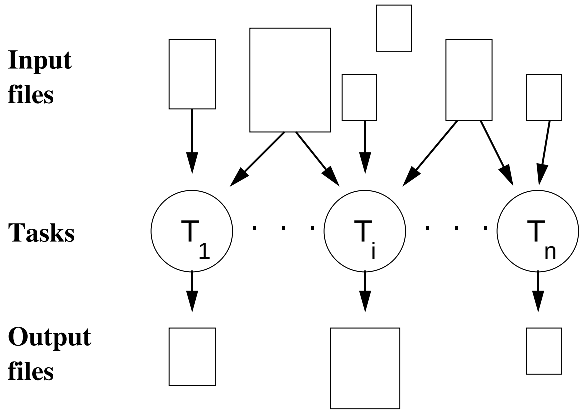

Figure 1. Application Model

MCell can be used to study the trajectories of neurotransmitters in the 3-D space between two cell membranes for different deformations of the membranes, where each deformation is described in a geometry file. Additional files of variable sizes are also needed for describing the initial locations of different molecules. The model described above is adequate for our purpose and should be general enough to accommodate other applications (e.g. general Monte-Carlo simulations).

MCell users and developers anticipate large-scale runs that contain tens of thousands of tasks with each task processing hundreds of MBytes of input and output data, with various task-file usage patterns. Furthermore, research work outside the scope of this paper addresses the question of *steerable* MCell runs when users can add new tasks on-the-fly, and modify the computational targets of existing tasks. Such runs will lead to fairly intricate task-file usage patterns and a model as general as the one we describe will be needed to study scheduling issues in the presence of computational steering.

2.2. Grid Model

We assume that the Computational Grid available to the user has the following *topology*: it is a set of k clusters of computing resources $\{C_j\}_{j=1,..,k}$ that are accessible to the user via k distinct network links. This is a logical topology, and this work does not attempt to take into account the actual physical network topology of the Grid. Our intent is to model a wide-area system, such as a *Worldwide Flock of Condors* [24] for instance. Each cluster contains a certain number of *hosts* where a host can be any computing platform, from a single-processor workstation to an MPP system, and is available for computation. From now on, we call

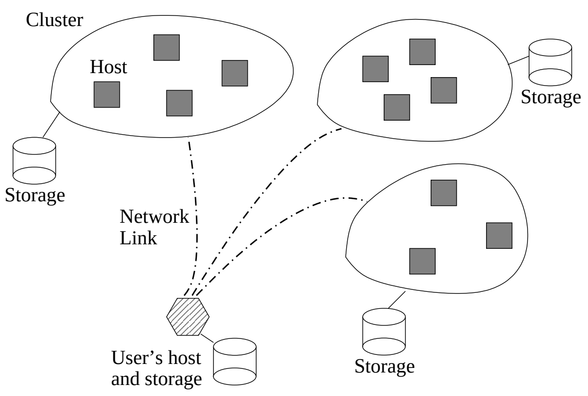

Figure 2. Environment Models

both hosts and network links *resources*. We do not impose any constraint on the performance characteristics of the resources and our simulator allows for arbitrary performance variation. The only requirement is that for each computation and file transfer, an estimate of running time is available. On interactive hosts, the estimate is the task *execution time* whereas on batch resources, it is the *turnaround time* (defined as the waiting time plus the execution time). Such estimates can be provided by the user, computed from analytical models or historical information, or provided by facilities such as the Network Weather Service (NWS) [31], ENV [28], Remos [19], Grid services such as those found in Globus [10], or computed from a combination of the above. Recent work shows that the accuracy of the performance estimates have an impact on the effectiveness of scheduling heuristics and that information about the probability distribution of the estimates should be exploited for scheduling [18, 26]. We explore scenarios for different levels of estimate accuracy in Section 5.

We assume that a storage facility (e.g. NFS, GASS [9], IBP [23]) is available at each cluster so that files can be shared among the processes running on different hosts in the cluster. For the first implementation of PST, we are planning to use IBP for storage management. Figure 2 shows an example of a Grid of a Grid with three clusters. In this work we assume that all input files are initially stored on the user's host, that all output files must be returned to this location, and that there are no inter-cluster file exchanges. We assume for now that once assigned, tasks do not migrate between resources. This scenario fits the current usage of several real-life, parameter sweep applications (e.g. MCell, INS2D [25]), and we leave alternate usage scenarios for future work. In this work we ignore possible storage constraints and assume unlimited storage space. Our model assumptions are discussed in the following section, but we believe that they make it possible to obtain initial meaningful results about a realistic environment while keeping the simulation tractable.

2.3. Model Discussion

Our Grid model makes several simplifying assumptions. Even though we allow network links to have arbitrary dynamic performance characteristics, we do not model network contention caused by the application itself. Instead we view the network as a set of distinct links emanating from the user's host and that can all be used in parallel. We believe that this assumption will need to be relaxed in future work. Since our purpose is not simulation per se, we will aim at using simulators developed by other research groups. For instance, the Micro-Grids simulator [15], when it becomes available, will allow us to precisely simulate network contention and study its impact on our current results.

Similarly, we do not take into account contention within a cluster for shared file access. Our justification is that wide-area file transfer cost dominate the cost of file access within the cluster, even in the presence of contention. While this is true in certain environments, it is certainly not general and we will need to enhance our own simulator so that it can simulate model contention for shared storage. For instance, this will be necessary to simulate high-bandwidth wide-area research networks such as the vBNS. At the moment we are planning to deploy the PST software on non-dedicated commercial wide-area networks with many clusters and we believe our simulation results will hold in those environments. The assumption of unlimited storage is realistic for current runs of MCell on our current testbed, but that assumption will be relaxed in future versions of our scheduling algorithm.

Our Grid model also assumes that there are no direct network links between clusters in the sense that file transfers cannot be performed by our scheduling algorithm between clusters. In other words, the only authoritative source of input files is the user's host. This prevents schedulers from making some optimizations when disseminating input files among the clusters. However, no heuristic we study in this paper is able to support such optimizations as this would require a considerably more precise understanding of the network. Our next step in this research will be to use a more complete network model and to consider any storage device for any file retrieval. This will allow not only for more flexible application scenarios, but also for the investigation of more sophisticated scheduling algorithms.

```
schedule() {
    (1)  compute the next scheduling event
    (2)  create a Gantt Chart, G
    (3)  foreach computation and file transfer currently underway
                compute an estimate of its completion time
                fill in the corresponding slots in G
    (4)  select a subset of the tasks that have not started execution: T
    (5)  until each host has been assigned enough work
              heuristically assign tasks to hosts (filling slots in G)
    (6)  convert G into a plan
}
```

Figure 3. Scheduling Algorithm Skeleton

3. Adaptive Scheduling for Parameter Sweeps

3.1. The Scheduling Algorithm

We call our scheduling algorithm `schedule()`. The general strategy is that it takes into account resource performance estimates to generate a *plan* for the assigning file transfers to network links and tasks to hosts. To account for the Grid's dynamic nature, `schedule()` can be called repeatedly so that the schedule can be modified and refined. We denote the points in time at which `schedule()` is called *scheduling events*, according to the terminology in [20]. We assume that at each scheduling event our scheduler has knowledge of: (i) the current topology of the Grid (number of clusters, number of hosts in those clusters, network and CPU loads), (ii) the number and location of copies of all input files, and (iii) the list of computations and file transfers currently underway or already completed.

Figure 3 shows the general skeleton for `schedule()` whose steps can be described as follows:

(1) determines the time of the next scheduling event. This can take into account environment behavior to increase or decrease the scheduling event frequency. A higher frequency means a higher adaptivity but also a higher scheduling cost.

(2) creates a Gantt chart [7], G, that will be used to keep track of task/host assignments. G contains as many columns as resources. Figure 4 shows an example of a Gantt chart for an environment containing two clusters with respectively two and three hosts.

(3) inserts slots corresponding to tasks that are currently running into the chart. Two examples are shown on Figure 4 as black-filled rectangular slots at the beginning of the chart (one file transfer and one computation).

(4) performs a task-space reduction that can be used to reduce `schedule()`'s execution time. This will be necessary for runs of real parameter sweep applications since we expect them to contain thousands of tasks.

(5) is the core of the algorithm, determining which task should be performed on which host. This step is detailed in Section 4. Examples of slot assignments are depicted on Figure 4 in gray. In this example, input file transfers are scheduled on the network link to cluster 2, the computation is then scheduled on a host within that cluster, and the output file is scheduled to be returned to the user's host.

(6) converts the Gantt chart into a *plan*, or a sequence of instructions. These instructions can then provide a schedule for deployment with Grid software services (for job submission and monitoring, data motion, etc.).

3.2. Discussion

Several steps of our scheduling algorithm can be implemented independently and this makes it possible to experiment with different techniques and strategies. Our ultimate goal is to instantiate the algorithm so that it is optimized for specific environments and applications. Furthermore, this instantiation should be

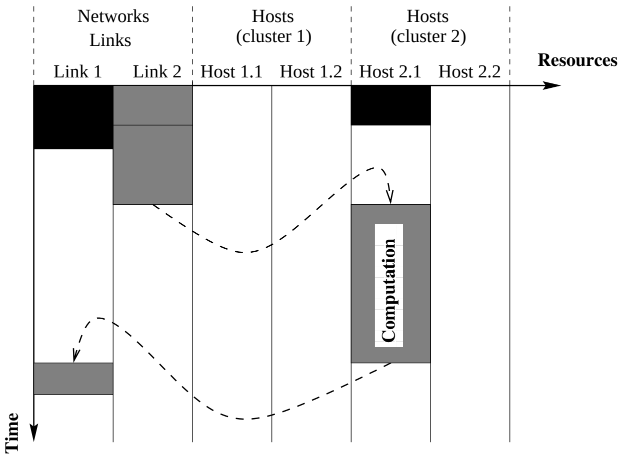

Figure 4. Sample Gantt chart

as dynamic and automatic as possible as the algorithm should be able to reconfigure itself on-the-fly to accommodate changing Grid conditions.

Step (1) allows for dynamic adjustment of the scheduling event frequency. A higher frequency leads to more adaptivity and should be better for unstable Grid environments. However, a higher frequency also means that `schedule()` is called more frequently. Depending on the computational cost of the scheduling, a high frequency might not be desirable. In the case of the PST software, a processor is usually dedicated to scheduling. Furthermore, given the granularity of the applications we are considering, there should be no need for very high frequencies. However, steps (3) and (5) might include remote access to Grid information services (e.g. NWS [31]) to perform performance prediction. It may then become necessary to reduce the scheduling event frequency because of latencies associated with Grid services. One can envision algorithms to dynamically tune the frequency in step (1). For instance, one could compute the deviation of the computation from what was planned during the previous call to `schedule()`. Large deviations suggest higher frequencies whereas low deviations suggest that the frequency can be decreased.

Step (3) obtains estimates for completion of ongoing file transfers and computations in order to start filling in slots in the Gantt chart. This is a little different from estimating just running times because more information is available. It is indeed conceivable that more precise forecasting techniques can be used because the required prediction is in the near future and because there are ways to compute percentage to completion. It may be that applications provides means to check on computation progress (this is however not the case for MCell). More generally, techniques using historical information from Grid services can lead to estimations of the percentage to completion. We have started experimenting with such techniques and will present results in a subsequent paper.

Step (5) in Figure 3 states that tasks are assigned to hosts until "enough" work has been assigned. Like step (4), this is intended to limit the time spent computing the schedule. Indeed, it makes little sense to assign tasks to hosts for times that are well beyond the next scheduling event since the schedule will be re-evaluated then. Since real runs will not be perfectly predictable, it is good practice to leave some margin of error and assign work until after the next scheduling event so that resources are utilized even if the performance predictions were pessimistic.

Step (6) processes the Gantt chart and transforms it into a set of task lists associated to each resource. The Grid infrastructure software in use is then responsible for sequencing file transfers and computations on the appropriate resources. Here there is some latitude for some choices concerning the actual implementation of the task sequencing. It may be that, due to unexpected performance misprediction, some resource cannot execute the next task on its list but could execute one of the subsequent ones. For instance, a file transfer for the output of a task that is unexpectedly lagging might cause a network link to stay unused. A solution is to relax the ordering of the list and allow subsequent file transfers to be performed immediately.

Our experience indicates that allowing output file transfers to be delayed until they can effectively occur is usually a good idea as it allows for better network bandwidth utilization while not disrupting the overall schedule to a great extent. This is the scheme used by `schedule()` in this paper. Further experiments would be required in order to investigate the trade-offs between resource utilization and schedule disruption.

Steps (1) and (5) use dynamic information about the status of the Grid resources and are key to the algorithm efficacy. **Our main focus in this paper is step (5) of the algorithm** and our results are presented in the following section. We also present preliminary experiments concerning step (1) in Section 5.

4. Performing Task/Host Assignment Decisions

4.1. Heuristics

We must identify heuristics that are applicable in Grid environments to perform assignment of file transfers to network links and of computations to hosts.

Moreover these heuristics must be reasonably computationally inexpensive with respect to the duration of a typical application task. Three simple heuristics for scheduling independent tasks for a uniform single-user environment are proposed in [17, 20]: *Min-min*, *Max-min*, and *Sufferage*. These three heuristics iteratively assign tasks to processors by considering all tasks not scheduled and computing Minimum Completion Times (MCTs). For each task, this is done by tentatively scheduling it to each resource, estimating the task's completion time, and computing the minimum completion time over all resources. For each task, a *metric* is computed using these MCTs, and the task with the "best" metric is assigned to the resource that lets it achieve its MCT. The process is then repeated until all tasks have been scheduled.

Min-min uses the Minimum MCT as a metric, meaning that the task that can be complete the earliest is given priority. The motivation behind Min-min is that assigning tasks to hosts that will execute them the fastest will lead to an overall reduced makespan. *Max-min*'s metric is the Maximum MCT. The expectation is to overlap long-running tasks with short-running ones. The rationale behind *Sufferage* is that a host should be assigned to the task that would "suffer" the most if not assigned to that host. For each task, its sufferage value is defined as the difference between its best MCT and its second-best MCT. Tasks with high sufferage value take precedence. Note that this definition of sufferage is a little different from the one presented in [20]. We found our definition easier to implement and experiments showed no differences between our version of sufferage and the one in [20]. We modified all three heuristics so that they (i) include input and output data transfer times when computing MCTs and (ii) take into account the fact that some files may already be present on remote storage devices. In addition, we implemented an extended version of the Sufferage heuristic: *XSufferage*.

In XSufferage the sufferage value is computed not with MCTs, but with *cluster-level MCTs*, i.e. by computing the minimum MCTs over all hosts in each cluster. Our first intuition was that Sufferage should be a nice way to exploit file locality issues without any a-priori analysis of the task-file dependence pattern. The idea is that if a file required by some task is already present at a remote cluster, that task would "suffer" if not assigned to a host in that cluster, provided the file is large compared to the available bandwidth on the cluster's network link. The sufferage value would then be a simple way of capturing such situations and ensuring maximum file re-use. This is somewhat reminiscent of the idea of *task/host affinities* introduced in [20], where some hosts are better for some tasks but not for others.

However that early experiments showed that the Sufferage heuristic as described above does not lead to makespans as good as the ones we expected. This can be explained easily. Assume that a task, say T_0, requires a large input file that is already stored on a remote cluster. If that cluster contains two (or more) hosts with nearly identical performance, which is often the case in practice, then both those hosts can achieve nearly the same MCT for that task. If the file is of significant size compared to network bandwidths available, then it is likely that those two hosts lead to the best and second-best MCTs for T_0. This means that the sufferage value will be close to zero, giving the task low priority. Other tasks may be scheduled in its place, generate load on the hosts in the cluster, and eventually force T_0 to be scheduled on some other cluster, thereby requiring an additional file transfer. This can have a dramatic impact on the overall application makespan as it leads to poor file re-use among tasks, especially in wide-area bandwidth-constrained environments.

We solved this problem in XSufferage by using a modified sufferage value definition. For each task and for each cluster we compute the task's MCT only for hosts in the given cluster and call that value the *cluster-level MCT*. The *cluster-level sufferage* value is computed as the difference between the best and second-best cluster-level MCT. The task with the highest cluster-level sufferage is given priority and is scheduled to the host that achieves the earliest MCT within the cluster that achieves the earliest cluster-level MCT. Appendix 8 gives formal descriptions of Max-min, Min-min, Sufferage, and XSufferage.

4.2. Simulating Parameter Sweeps in Grid Environments

In order to evaluate the efficacy of the heuristics described earlier we developed a Grid parameter sweep simulator. At present, little software is available for Grid simulation. Among the most promising work, the Bricks project [30] addresses the question of simulating heterogeneous distributed environment for the purpose of evaluating scheduling strategies, but no public implementation is available at the moment. Furthermore, Bricks targets "global computing systems" [5, 27, 11, 10] rather than application schedulers. It assumes constant task and data arrival rates to servers and uses queuing theory in an attempt to model many users who asynchronously interact with a global computing system. By contrast, our simulator is purely event-driven which is more appropriate in our framework where the scheduler knows all tasks a-priori and is in charge of only one application. The Micro-Grids [15] project will also be of interest for gaining insight on how our simulation results hold under more realistic assumptions. At present, it cannot be used to perform large numbers of runs of large-scale applications as it emulates the Grid rather than simulates runs of the application. However, Micro-Grids uses a network simulator that could help us model network traffic more accurately by taking into account physical network topology and link contention due to that topology.

Our simulator allows us to compare heuristics under the same load conditions, in a reproducible manner, for a wide variety of system states and application scenarios. In addition, we verified the accuracy of our simulated results by comparing experimental runs in shared, production environments with similarly loaded simulation application execution times. Our simulator takes as input `schedule()`, a task/host assignment heuristic, a description of the application tasks and input/output files, and a description of the Grid topology with performance characteristics of Grid resources. These characteristics can be constant values, samples from random distributions, or traces from the NWS [31]. In this work we use only NWS traces as they lead to more realistic models. The simulator also allows for adding and removing resources dynamically, but we do not perform any experiments with transient resources in this paper. The output of the simulator is a makespan value based on the set of input parameters. More details on the simulator can be found in [6].

4.3. Simulation Results

4.3.1. Random Grids and Applications

In order to perform a fair comparison of task/host selection heuristics we generated 1000 simulated Grids and 2000 simulated applications. We then randomly picked Grid/application pairs among the 2,000,000 possible, and ran our simulator for each pair with all heuristics. The simulations in this section assume 100% accurate performance estimation and scheduling events occur every 500 seconds. The expectation is that computing statistical characteristics of makespans achieved by each heuristic is representative if the sample size is large enough, that is if enough Grid/application pairs are simulated. Before presenting the results, let us describe how Grids and applications were generated.

In what follows we denote by $U(x,y)$ the discrete integer uniform probability distribution on the interval $[x,y]$ where x and y are integers. Each Grid contains a $U(2,12)$ number of clusters and each of those clusters contains a $U(2,32)$ number of hosts. The performance of each host is modeled by a CPU load trace randomly picked among 50 different actual traces obtained from the NWS for various hosts. Each trace is then shifted by a random offset, so that two hosts using the same CPU trace do not exhibit the same behavior at the same time. Similarly network link performance is modeled by randomly picking latency/bandwidth traces among 20 different NWS traces. All the NWS traces that we use for simulations in this paper typically span 4 days of real time and were obtained for hosts in various US research institutions and for network links between these institutions (commercial Internet or vBNS). Traces were collected during the first week of November 1999.

In accordance with typical MCell scenarios we generate applications as sets of independent Monte-Carlo simulations, with the tasks of a simulation sharing a (potentially large) input data file for describing 3-D geometries. All tasks take as input one additional file of 1 KByte and generate an output file of 10 KBytes. An application is composed of a $U(2,10)$ number of Monte-Carlo simulations, with each simulation composed of a $U(20,1000)$ number of tasks. Each task requires a $U(100,300)$ number of seconds of computation on an unloaded base CPU. Finally, the size in KBytes of the geometry file associated with each Monte-Carlo simulation is $U(400,100000)$, meaning that those files can reach the size of approximatively 95 MBytes. These distributions are representative of what can be expected from real MCell applications. We generated one-thousand applications following exactly this method, and we also generated another thousand by adding random file/task dependencies. The idea is to create some perturbation of the regular structure of the file/task dependency graph and investigate if such a perturbation has an impact on the relative performances of the different task/host selection heuristics. The perturbation consisted in adding a number of random additional dependencies on the order of one fifth of the total number of tasks. Even though such perturbations take us away from typical MCell applications, it can be interesting to see how they affect the heuristics.

The results are summarized in Table 4.3 for both standard applications and applications with file/task dependencies perturbation. For each scheduling heuristic (including a self-scheduled workqueue [12]) and for each type of application, the table contains three performance values. The results are computed over 1000 random Grid/application pairs. We use the geometric mean of the makespans rather than the arithmetic mean to account for the fact that, depending on the Grid and the application, makespans in the sample

Table 1. Results for Random Grid/Application pairs

Scheduling	No Perturbation			Perturbation		
	Geometric Mean (sec)	Av. Degradation from Best (%)	Average Rank	Geometric Mean (sec)	Av. Degradation from Best (%)	Average Rank
Max-min	2390	17.3	3.1	2549	18.9	3.1
Min-min	2452	21.2	3.0	2619	23.2	3.0
Sufferage	2329	14.1	2.8	2505	16.7	2.9
XSufferage	2174	6.2	1.8	2316	7.9	1.8
Workqueue	2850	42.2	4.3	3091	48.1	4.2

space can be of different orders of magnitude (one large makespan could easily dominate the arithmetic mean).

The second performance value is the "average degradation from the best" which is a measure of how far a heuristic is from the best heuristic on average. For each heuristic, it is computed as the arithmetic mean over all Grid/application pairs of the relative difference of the makespans for that heuristic and for the best heuristic. The smaller that value, the closer the heuristic to being the best one on average.

The third performance value is the "average rank" of a heuristic over all Grid/application pairs. The average rank is computed as the arithmetic mean of the rank (1 to 5), where the heuristic leading to the best makespan is of rank 1 and the one leading to the worst makespan is of rank 5.

These three different performance values all have slightly different interpretations and make it possible to gain a clear understanding of how the heuristics compare with one another. For instance, it could be that heuristic 1 is the best one in most cases, and that heuristic 2 is the second-best one in most cases as well. In that case, their average ranks will be close to 1 and 2 respectively. This might lead us to think that heuristic one is preferable. However, it is possible that the average degradation from best are respectively 20% and 5%. For instance, it can be that when heuristic 1 is not the best one it is far worse than the best one, whereas heuristic 2 might not be best often, but is never far behind the best one in practice. In that case, we would probably conclude that heuristic 2 is preferable.

The main message from Table 4.3 is that XSufferage is the best heuristic as it leads to the best geometric mean, average degradation from the best, and average rank for perturbed and non-perturbed applications. Its average degradation from the best is at least twice smaller than that of any other heuristics for standard applications and applications with perturbation. Its average rank is better than any other by 1 unit. Note that the workqueue, in these experiments, is the least efficient scheduling algorithm as its average rank is larger than 4 units. Max-min and Sufferage are comparable with a slight advantage to Max-min, and Min-min seems less efficient.

Note that all the results in Table 4.3 are averages over a large number of experiments. In the following sections we will see cases where Min-min leads to good makespans when compared to Max-min and Sufferage. We claim that the experimental results presented in this section are sufficient to show that XSufferage is the one of the four heuristics that leads to best schedules for parameter sweep applications, given the models described in Section 2.

4.3.2. Varying Shared File Sizes

Figure 5(a) show simulation results for the following application and Grid. The application consists of 1600 tasks, where each task takes as input a 10K un-shared file and one of eight identical shared files, each shared by 200 tasks. All tasks are identical in terms of computational cost (200 seconds on an unloaded base CPU) and produce a 10K output file. This application setting is comparable to what some of our target parameter sweep applications require. We simulate a Grid such as one that could realistically be used by a user based at UCSD. That Grid contains 5 clusters containing respectively 6, 6, 8, 20, and 20 hosts. The performance characteristics of the hosts are based on actual CPU traces obtained via the Network Weather Service. The network links are also modeled from NWS traces obtained over the course of a day between a workstation at UCSD and several remote sites accessible by commercial Internet links or the vBNS. Bandwidths on these links varies from as little as 6 KBytes/sec to 600 KBytes/sec depending on the link and on the time of the day. In terms of bandwidth averages, one can classify two of the links as *fast* (500 KBytes/sec), three of the links as *moderate* (between 100 and 200 KBytes/sec), and one as *slow* (50 KBytes/sec). One

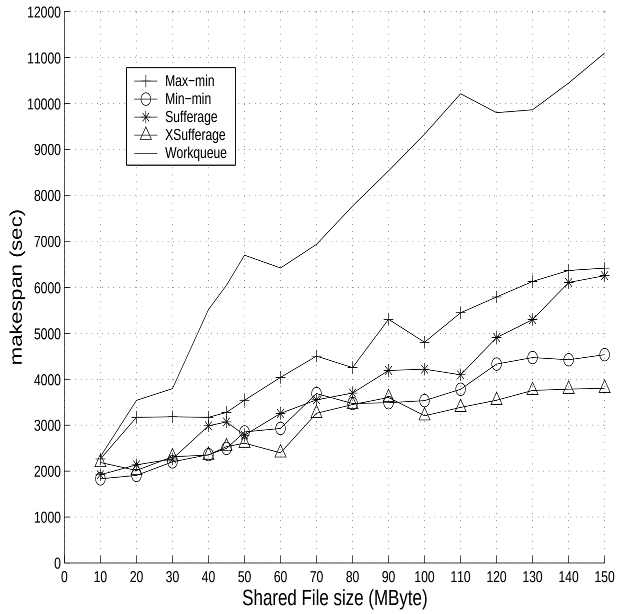

Figure 5. Makespan vs. shared file size for different heuristics

of the two large 20-host clusters is accessible via a fast link. For large shared file sizes on the graphs, the average ratio between file transfer time and computation time for one task is about 3 on a fast link and 30 on a slow link. For the smallest shared file sizes in our study, that ratio is about 0.2 on a fast link and 2 on a slow link. For these experiments, the interval between scheduling events was always 500 seconds, and we assumed 100% accurate performance estimations.

The graph in Figure 5 plots makespans vs. shared file shared file sizes between 10 and 150 MBytes for the four heuristics and the self-scheduled workqueue. For very small shared file sizes (up to 100 KBytes, not plotted on the graph), the workqueue leads to better makespans than other heuristics when files are so small that the effect of file sharing becomes negligible. However, the workqueue quickly becomes inefficient when shared file size increases. The other heuristics perform similarly for small shared file sizes but one can see on the graph that XSufferage performs at least about 20% better than Min-min and 40% better than Max-min and Sufferage for a file size of 150 MBytes. We obtained similar results with different Grid configurations. We also performed experiments with much larger file sizes. Even though those experiments are not very realistic given the current networking capabilities they provide information about what happens when file re-use is the only constraint for achieving good scheduling. The results showed that XSufferage constantly outperforms all other heuristics by at least 50%. These results show that XSufferage does a better job at capturing and taking advantage of file sharing patterns to maximize file re-use.

5. Adaptive Scheduling

5.1. Quality of Information

A new avenue of research that we are beginning to explore is the study of *Quality of Information* (QoI) on scheduling, that is the impact of the performance estimation accuracy and other qualitative attributes on different scheduling strategies . We expect different heuristics to react differently to degrading levels of accuracy and that strategies that do not depend on performance estimation and forecast (e.g. self-scheduled ones [16]) will be more performance-efficient when QoI is low. Low QoI can also be accounted for in adaptive scheduling algorithms such as `schedule()`. The following section presents our first simulation results for different levels of QoI and for increasingly adaptive versions of `schedule()`.

Our initial model for simulating different levels of QoI is simple. Our simulator allows us to obtain 100% accurate estimates for all file transfer or computational times and we add random noise to those estimates to simulate inaccurate performance estimates. For each estimate used by the scheduling algorithm we introduce a percentage error that is uniformly distributed on the interval $[-p, +p]$ where p is a value between 0% and 100%. Perfectly accurate QoI corresponds to $p = 0$, whereas $p = 10$ means that every 100% accurate estimate will be randomly increased or decreased by up to 10%.

This model is sufficient for obtaining initial results concerning the impact of QoI on the scheduling of parameter sweep applications, however it makes two assumptions that are not realistic for real forecasting services that will be deployed in Computational Grids. First, it assumes that QoI behavior is the same for all estimates (for file transfer times and computational times) and for all resources. This is clearly not the case as network behavior is significantly different from CPU behavior for performance prediction purposes [32], and some resources will generally be more predictable than others on a regular basis. Second, it assumes that QoI behavior does not depend on whether a forecast

The term "quality of Information is used to describe qualitative aspects of performance predictions in the Performance Prediction Engineering Project [14].

is needed for an event in the short-term or in the long-term. For instance, our model uses the same error model for predicting a file transfer time if the transfer is initiated in the next minute or in an hour. A more realistic model should probably try to capture some *decay* of the QoI as predictions become more and more long-term. Note that this issue becomes less critical for high scheduling event frequencies. Future work will aim at providing a more realistic model of QoI based on experiments with deployed Grid services, such as the Network Weather Service [31], and with a variety of Grid resources.

5.2. Simulation Results

Figure 6 shows simulation results for four different scheduling event frequencies and decreasing QoI levels.

We use the same simulated Grid as the one used in Section 4.3.2 and the application is modeled after an MCell computation that performs eight moderate-size Monte-Carlo simulations (100 tasks each) for eight different geometry configurations of a neuro-muscular junction. Geometry files are on the order of 40 MBytes, meaning that network transfer times for those files take on average 80 seconds on a fast link and about 800 seconds on a slow link. The average task computational time over all hosts is about 110 seconds, can be as fast at 90 seconds, and as slow as 350 seconds depending on the host and on its load when the task is running (as simulated by an offset in an NWS CPU load trace).

All data points in the graphs of Figure 6 are computed as the average makespan over 50 simulated runs. This is necessary since we introduce random noise to performance estimates in order to simulate different levels of QoI. All graphs plot average makespans vs. values of p (defined in Section 5.1, for the heuristics presented in Section 4.1 as well as for a self-scheduled workqueue algorithm. Since the workqueue does not make use of performance estimates it is not sensitive to QoI. It is shown as a horizontal solid line on the four graphs (with a makespan of 1730 seconds). The variances associated with the 50 samples for each data point were small for all heuristics: coefficients of variations were on the order of 5%.

Graph 6(a) plots results when there is only one initial scheduling event, meaning that the scheduling algorithm is not adaptive. All heuristics but Max-min lead to better makespans than the workqueue for perfect QoI ($p = 0$), but their performance degrades very rapidly when p increases. Max-min leads to the worst makespans, but over all, all heuristics lead to makespans at least 40% larger than the workqueue when p is greater than 50. This result is not surprising as the cumulative errors of performance estimates impact the computations of the various MCTs required by the heuristics.

Graph 6(b) shows the results when there is a scheduling event every 500 seconds, or 3 times during each run of the application in this case. One can notice that some heuristics outperform the workqueue for values of p up to 20. Max-min and Sufferage exhibit less performance as soon as the QoI is not perfect.

Graph 6(c) shows the results when there is a scheduling event every 250 seconds, or between 5 and 8 times for each run depending on the heuristic being used. The effect observed on graph 6(b) is more pronounced in that heuristics become more tolerant to low QoI thanks to increased adaptivity, even though Max-min still leads to large makespans. Sufferage outperforms the workqueue for values of p lower than 30, whereas Min-min and XSufferage lead to better makespan than the workqueue for all values of p. For perfect accuracy, XSufferage outperforms workqueue by as much as 25%.

Finally, Graph 6(d) shows results for scheduling events every 125 seconds, or between 11 and 14 times per run. Sufferage now outperforms the workqueue for p up to 80, whereas Min-min and XSufferage keep benefiting from increased adaptivity. The results show little improvements for higher scheduling frequencies. This is due to the granularity of the application: since tasks take at least 90 seconds, little can be gained by calling `schedule()` more than once every 125 seconds. For "good" QoI ($p < 5\%$), XSufferage always outperforms Min-min.

Note that these results are preliminary and that it is difficult to use them to rank the different heuristics according to their respective robustness to inaccurate performance predictions. It will be necessary to perform experiments for large numbers of different Grid configurations and application structures as was done in Section 4.3.1. A future paper will contain results from such experiments as well as a more in-depth study of QoI issues.

Note also that in these experiments we assume that the QoI does not depend on the scheduling event frequency (see the discussion in Section 5.1). However, a high scheduling frequency implies that the heuristics do not use long-term predictions (see the discussion on step (5) in Section 3.2). Assuming that short-term predictions are typically more accurate than long-term predictions, higher scheduling frequencies lead to improved QoI. On Graph 6(d) we show simulation results for values of p up to 100, but we expect that in reasonably stable Grid environments with appropriate forecasting services, the performance estimation error will not be as large as +/- 100% for short-term predictions.

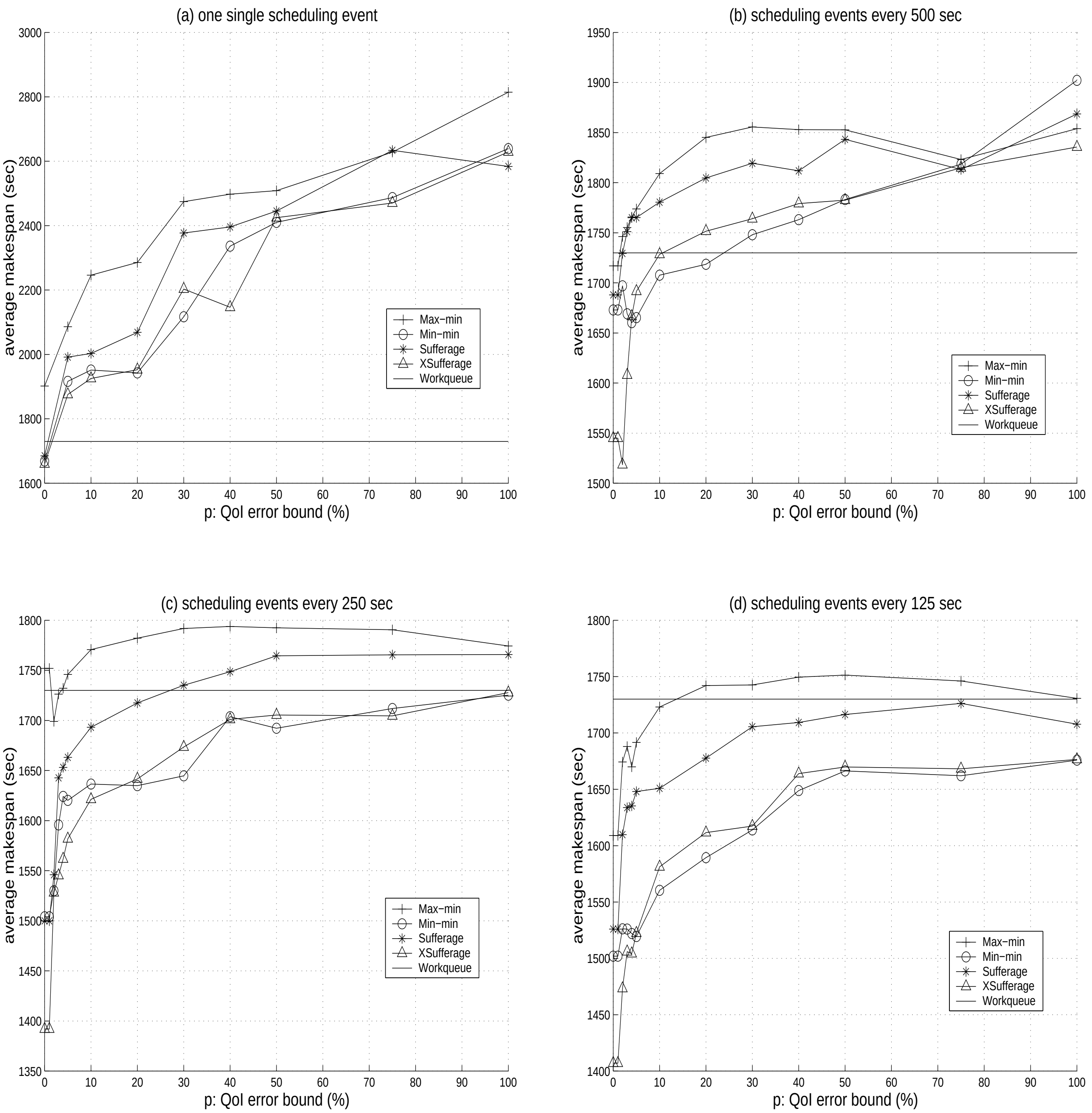

Figure 6. Simulation results for various levels of QoI and adaptivity

The left part of the graph should be more representative of what we can expect for real systems.

Finally, the scheduling event frequency impacts the performance of the adaptive algorithm even for perfect QoI. For instance, XSufferage has an average makespan around 1550 seconds for $p = 0$ and scheduling events every 500 seconds, whereas that average makespan becomes lower than 1400 seconds for larger frequencies. All heuristics but Max-min achieve better makespan for higher frequencies in the case of perfect QoI. This is rather counter-intuitive as one would expect that perfect QoI would not mandate any adaptivity at all. The fact is that applying those heuristics on small sets of tasks leads to better scheduling decisions and shorter makespans. As tasks complete, successive calls to `schedule()` apply the heuristics to the decreasing sets of tasks, leading to better overall makespans. This suggests that calling the scheduling algorithm repeatedly is a good idea for Sufferage, Min-min, and XSufferage, even if the environment is very predictable.

6. Summary of Simulation Results

The simulation results in Section 4.3.1 showed that XSufferage is on average a better heuristic than the traditional Max-min, Min-min, Sufferage, and self-scheduled workqueue for scheduling parameter sweep applications such as MCell on Computational Grids, provided the models of Section 2 hold. We believe that Xsufferage leads to better results because it better captures the file/task dependencies of the application and leads to improved file re-use. This claim is supported by the results in Section 4.3.2. Indeed, all experiments we have conducted with very large shared files seem to indicate that XSufferage leads to better makespans than its contenders. Finally, the preliminary study of Quality of Information and adaptivity in Section 5 showed that all heuristics can benefit from increased adaptivity and from increased QoI. The results seem to indicate that XSufferage leads to very good results for good QoI and compares well with other heuristics for poor QoI.

It is always difficult to make general statements about the relative efficiency of scheduling algorithms since the space of possible Grid configurations and application structures is very large. The solution is to sample both the Grid and the application space as much as possible as was done in Section 4.3.1. Future work will contain such sampling for experiments similar to the ones presented in Section 4.3.2 and 5 in order to make the results concerning shared file sizes and QoI more general. Ironically, performing such large-scale simulations in the Grid/application space is itself a parameter sweep application and we will probably use the PST software to distribute it on a real computational Grid.

7. Related Work

A large number of research papers address the question of mapping sets of tasks onto sets of processors in a view to minimizing overall execution time. Many of these papers address the case where tasks are independent [17, 12, 16, 20]. Scheduling heuristics found in [20] were adapted to our framework as discussed in Section 4. All these papers make simplifying assumptions for task execution times (constant, following a truncated Gaussian distribution, etc.) and none of them take into account data storage issues.

The work described in [2] focuses on scheduling applications structured as DAGs on heterogeneous sets of processors and uses heuristics that are related to Max-min and Min-min for *Level-by-Level* scheduling of the graphs. However, special attention is paid to data storage issues which makes that work related to the research presented in this paper. A major difference between our work and the development in [2] is that the latter assumes constant perfectly predictable performance characteristics for resources as should be available in advanced reservation QoS environments. Also, the different application structures (Parameter Sweep vs. DAGs) lead to many differences between the models in this paper and in [2]. For instance, data repositories are located anywhere on the network (as opposed within a cluster) and datasets are pre-staged to these repositories. Finally, [4] contains a survey that encompasses several heuristics in addition to the ones described in [20]. We will consider these heuristics in our future work. This work contrasts to others in that we: (i) take into account application data storage; (ii) model shared, heterogeneous computational and network resources with realistic dynamic performance characteristics; (iii) study the impact of the accuracy of performance prediction; (iv) introduce a new heuristic for scheduling parameter sweep applications (XSufferage).

This work is also related to our work on an AppLeS Parameter Sweep Template (PST) in that the results in this paper provide a good justification that XSufferage should be implemented as part of the PST scheduler. PST will provide with a practical way to deploy and schedule parameter sweep on the computational Grid using available software infrastructures and will be described in a future paper. PST itself is related to the Nimrod project [1]. Nimrod targets parameter sweep applications but its scheduling approach is different from ours as it is based on deadlines and on a Grid economy model. Also, to the best of our knowledge, Nimrod does not take into account dynamic Grid conditions or file locality constraints for scheduling. In fact, the work in this paper and our work on the PST software should be applicable to Nimrod and one can envision an implementation of PST as a Nimrod module.

8. Conclusion and Future Work

In this paper we have proposed an adaptive scheduling algorithm for parameter sweep applications in Grid environments. In particular, we address the case of applications where tasks can share input files (e.g. MCell [29]) and the case of non-dedicated Computational Grids that span non-dedicated wide-area networks. After precisely defining our application and Grid model, we adapted three standard heuristics for performing task/host assignment (*Max-min*, *Min-min*, *Sufferage*) and proposed an extension of the *Sufferage* heuristic, *XSufferage*. We also introduced the notion of *Quality of Information* (QoI) to account for inaccuracies in performance predictions. We use simulation to compare the four heuristics and a self-scheduled workqueue algorithm in multiple settings with various shared files sizes, levels of QoI, application structures, Grid topologies and resources. The simulation results demonstrated that: (i) XSufferage leads to better schedules on average by quite a large margin; (ii) Increased adaptivity benefits all four heuristics even for perfect QoI; (iii) XSufferage leads to better schedules for larger shared file sizes; (iv) XSufferage is as tolerant as the other heuristics to poor QoI and more efficient for good QoI.

Future work will provide improvements to our models such as more realistic network and storage models (encompassing shared-storage and link contention, and limited storage space), and alternate application usage scenarios. We will also study the concept of QoI further by investigating realistic QoI models and performing more QoI-related experiments with our simulator. New heuristics such as the ones found in [4, 21] will considered for implementing step (5) of our algorithm. As discussed in Section 3.2, all steps of the algorithm can lead to new research in different directions (performance prediction and forecasting, task-space reduction, trade-offs between schedule disruption vs. maximum resource utilization). Also, the algorithm can be adapted to provide ways to perform adaptive scheduling for other classes of applications by using different heuristics in step (5). Finally, we will incorporate the results here, as well as many of these future improvements, into a practical programming environment and adaptive scheduler for parameter sweep applications on the Grid, an AppLeS Parameter Sweep Template. We believe that such software will provide a useful first step in achieving performance and programmability for applications in Grid environments.

Acknowledgements

The authors would like to thank the reviewers for their insightful comments as well as members of the AppLeS group for their help.

Appendix A: Task/host Selection Heuristics

Let $H_{j,k}$ denote the k^{th} host within the j^{th} cluster and $C(T_i, H_{j,k})$ the estimated completion time of task T_i on host $H_{j,k}$. Let us define the *argmin* operator:

Definition: Given a function f from $\mathbb{R}^n$ into $\mathbb{R}$,

$$f(\text{argmin}_{x \in \mathbb{R}^n} f(x)) = \min_{x \in \mathbb{R}^n} f(x).$$

The operator denotes one of the possible vectors that achieves the minimum of the function f. The way ties are broken is left the implementation and in this work they are broken randomly. An *argmax* operator can be defined in a similar fashion. Assuming a task set T, we can now describe each heuristic as follows:

Min-min

while $(T \neq \varnothing)$
    foreach $(T_i \in T)$
        $(c_i^{(1)}, h_i^{(1)}) = \text{argmin}_{j,k}(C(T_i, H_{j,k}))$
    end foreach
    $s = \text{argmin}_i(C(T_i, H_{c_i^{(1)}, h_i^{(1)}}))$
    assign T_s to $H_{c_s^{(1)}, h_s^{(1)}}$
    $T = T - \{T_s\}$
end while

Max-min

while $(T \neq \varnothing)$
    foreach $(T_i \in T)$
        $(c_i^{(1)}, h_i^{(1)}) = \text{argmin}_{j,k}(C(T_i, H_{j,k}))$
    end foreach
    $s = \text{argmax}_i(C(T_i, H_{c_i^{(1)}, h_i^{(1)}}))$
    assign T_s to $H_{c_s^{(1)}, h_s^{(1)}}$
    $T = T - \{T_s\}$
end while

Sufferage

while $(T \neq \varnothing)$
  foreach $(T_i \in T)$
    $(c_i^{(1)}, h_i^{(1)}) = \text{argmin}_{j,k}(C(T_i, H_{j,k}))$
    $(c_i^{(2)}, h_i^{(2)}) = \text{argmin}_{j \neq c_i^{(1)}, k \neq h_i^{(1)}}(C(T_i, H_{j,k}))$
    $suf_i = C(T_i, H_{c_i^{(2)}, h_i^{(2)}}) - C(T_i, H_{c_i^{(1)}, h_i^{(1)}})$
  end foreach
  $s = \text{argmax}_i(suf_i)$
  assign T_s to $H_{c_s^{(1)}, h_s^{(1)}}$
  $T = T - \{T_s\}$
end while

XSufferage

while $(T \neq \varnothing)$
  foreach $(T_i \in T)$
    foreach cluster j
      $h_{i,j} = \text{argmin}_k(C(T_i, H_{j,k}))$
    end foreach
    $c_i^{(1)} = \text{argmin}_j(C(T_i, H_{j,h_{i,j}}))$
    $c_i^{(2)} = \text{argmin}_{j \neq c_i^{(1)}}(C(T_i, H_{j,h_{i,j}}))$
    $suf_i = C(T_i, H_{c_i^{(2)}, h_{i,c_i^{(2)}}}) - C(T_i, H_{c_i^{(1)}, h_{i,c_i^{(1)}}})$
  end foreach
  $s = \text{argmax}_i(suf_i)$
  assign T_s to $H_{c_i^{(1)}, h_{i,c_i^{(1)}}}$
  $T = T - \{T_s\}$
end while

References

[1] D. Abramson and J. Giddy. Scheduling Large Parametric Modelling Experiments on a Distributed Metacomputer. In *PCW'97*, Sep. 1997.

[2] A. Alhusaini, V. Prasanna, and C. Raghavendra. A Unified Resource Scheduling Framework for Heterogeneous Computing Environments. In *Proceedings of the 8th IEEE Heterogeneous Computing Workshop (HCW'99)*, pages 156–165, Apr. 1999.

[3] F. Berman. *The Grid, Blueprint for a New computing Infrastructure*, chapter 12. Morgan Kaufmann Publishers, Inc., 1998. Edited by Ian Foster and Carl Kesselman.

[4] R. Braun, H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW'99)*, pages 15–29, Apr. 1999.

[5] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 1997.

[6] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments. Technical Report RR1999-46, École Normale Supérieure de Lyon, France, Sep. 1999.

[7] W. Clark. *The Gantt chart.* Pitman and Sons, London, 3rd edition, 1952.

[8] I. Foster and C. Kesselman, editors. *The Grid, Blueprint for a New computing Infrastructure.* Morgan Kaufmann Publishers, Inc., San Francisco, USA, 1998.

[9] I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sicth workshop on I/O in Parallel and Distributed Systems*, May 1999.

[10] I. Foster and K. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[11] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. In *IEEE Computer 32(5)*, volume 32(5), May 1999. page 29-37.

[12] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47:185–197, 1997.

[13] http://apples.ucsd.edu.

[14] http://apples.ucsd.edu/perf.html.

[15] http://www csag.ucsd.edu/projects/grid/microgrid.html.

[16] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 318–328, June 1996.

[17] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonindentical processors. *Journal of the ACM*, 24(2):280–289, Apr. 1977.

[18] T. Kidd and D. Hensgen. Why the Mean is Inadequate for Accurate Scheduling Decisions. In *Proceedings of the 3rd International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN'99)*, Jun. 1999.

[19] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A Resource Query Interface for Network-Aware Applications. In *Proceedings of the 7th IEEE Smposium on High-Performance Distributed Computing*, July 1998.

[20] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *8th Heterogeneous Computing Workshop (HCW'99)*, Apr. 1999.

[21] M. Mitzenmacher. How useful is old information. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 83–91, 1997.

[22] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[23] J. Plank, M. Beck, W. Elwasif, T. Moore, , M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the Network. In *Proceedings of NetSore'99: Network Storage Symposium, Internet2*, 199.

[24] J. Pruyne and M. Livny. A Worldwide Flock of Condors : Load Sharing among Workstation Clusters . *Journal on Future Generations of Computer Systems*, 12, 1996.

[25] S. Rogers and D. Ywak. Steady and Unsteady Solutions of the Incompressible Navier-Stokes Equations. *AIAA Journal*, 29(4):603–610, Apr. 1991.

[26] J. Schopf and F. Berman. Stochastic Scheduling . In *Proceedings of SuperComputing'99, Portland*, 1999.

[27] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf : Network based Information Library for Globally High Performance Computing. In *Proc. of Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe*, pages 39–48, February 1996.

[28] G. Shao, F. Breman, and R. Wolski. Using Effective Network Views to Promote Distributed Application Performance. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[29] J. Stiles, T. Bartol, E. Salpeter, , and M. Salpeter. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. *Computational Neuroscience*, pages 279–284, 1998.

[30] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, pages 97–104, Aug 1999.

[31] R. Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. In *6th High-Performance Distributed Computing Conference*, pages 316–325, August 1997.

[32] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU Availability of Time-shared Unix Systems on the computational Grid. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, Aug 1999.

Henri Casanova is a Computer Science and Engineering project scientist at the University of California, San Diego. His research interests include all areas of metacomputing, and in particular theoretical models for the efficient scheduling of distributed application in computational Grid environments. He received his BS from the École Nationale Supérieure d'Électrotechnique, d'Électronique, d'Informatique et d'Hydraulique de Toulouse (ENSEEIHT), his MS from the Université Paul Sabatier, Toulouse, and his PhD from the University of Tennessee, Knoxville.

Arnaud Legrand is a Computer Science graduate student at the École Normale Supérieure, the leading French scientific research and teaching institution, Lyon, France. He is interested in parallelism, metacomputing and numerical simulation and is currently working at the Laboratoire de l'Informatique et du Parallélisme (Parallel Computing Laboratory, ENS Lyon) on linear algebra algorithms that are taylored to heterogeneous environments.

Dmitrii Zagorodnov is currently pursuing a PhD at the Department of Computer Science and Engineering at the University of California, San Diego. He has received B.S. and M.S. degrees in computer science from the University of Alaska Fairbanks in 1995 and 1997, respectively. His current research interest is in fault tolerance for distributed systems.

Francine Berman is a Professor of Computer Science and Engineering at U. C. San Diego, Senior Fellow at the San Diego Supercomputer Center, Fellow of the ACM, and founder of the Parallel Computation Laboratory at UCSD. Her research interests over the last two decades have focused on parallel and distributed computation, and in particular the areas of programming environments, tools, and models that support high-performance computing. She received her B.A. from the University of California, Los Angeles, and her M.S. and Ph.D. from the University of Washington.