

Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments

Francine Berman, Henri Casanova, Dimitrii Zagarodnov, Arnaud Legrand

▶ **To cite this version:**

Francine Berman, Henri Casanova, Dimitrii Zagarodnov, Arnaud Legrand. Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments. [Research Report] LIP RR-1999-46, Laboratoire de l'informatique du parallélisme. 1999, 2+40p. hal-02101774

HAL Id: hal-02101774

https://hal-lara.archives-ouvertes.fr/hal-02101774

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire de l'Informatique du Parallélisme



École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON
n° 5668



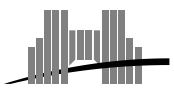


Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments

Francine Berman, Henri Casanova, Dmitrii Zagorodnov
(Department of Computer Science and Engineering, University of California San Diego, USA)
Arnaud Legrand
(École Normale Supérieure de Lyon, France)

September 1999

Research Report N° 99-46



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr



Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments

Francine Berman, Henri Casanova, Dmitrii Zagorodnov
(Department of Computer Science and Engineering,
University of California San Diego, USA)
Arnaud Legrand
(École Normale Supérieure de Lyon, France)

September 1999

Abstract

Fast networks have made it possible to aggregate distributed CPU, memory, and storage resources into Grids that can deliver considerable performance. However, achieving performance on such systems requires good performance prediction which is usually difficult due to their dynamic and heterogeneous nature. This is especially true for parallel applications whose performance is highly dependent upon the efficient coordination of their constituent components (e.g. computation and data).
The goal of the AppLeS project is to develop application-level scheduling agents that provide mechanisms for automatically scheduling individual applications on production heterogeneous systems. AppLeS agents utilize the Network Weather Service (NWS) to monitor and forecast the varying performance of resources potentially usable by their applications. Each AppLeS uses static and dynamic application and system information to select viable resource configurations and evaluate their potential performance. The AppLeS then interacts with the appropriate resource management system to implement the application's network transfers and computational tasks.
The next generation of AppLeS agents aims at providing *templates* that can be used for scheduling classes of structurally similar applications. In this document we introduce a template for scheduling Parameter Sweep applications (application consisting of *large* number of independent tasks, with possible input data sharing). We have designed a general scheduling algorithm that can adapt to Grid environments and use a variety of strategies and heuristics to assign tasks and data to resources. In order to evaluate and compare those heuristics we have built a simulator as part of the template. The simulator makes it possible to rapidly conduct large numbers of experiments in a variety of environments. Our starting point was to use widely accepted heuristics that have been proposed in the litterature and venture improvements given our Grid and application model. This document presents the implementation of our simulator and explains how it will be used to obtain new research results in the field of Grid scheduling.

Keywords: Scheduling, Meta-computing, Grid-computing, Grid, Parameter-Sweep Template, Sufferage, File-sharing

Résumé

Les réseaux haut débit ont permis de connecter des capacités de calcul et de stockage distribuées en des *grilles* susceptibles de fournir des puissances de calcul considérables. Cependant, l'utilisation efficace de tels systèmes recquiert de bonnes prédictions des performances des différentes ressources, ce qui est généralement difficile en raison de l'hétérogénéité et du caractère dynamique de ces environnements. Ce constat est tout particulièrement vrai dans le cas d'applications parallèles dont l'efficacité dépend de la bonne coordination de ses différents composants.
L'objectif du projet AppLeS est de développer des agents chargés de l'ordonnancement dans des environnement de calculs distribués hétérogènes et spécialisés pour chaque application. Les agents AppLeS utilisent le *Network Weather Service* (NWS) pour contrôler et prédire les performances des ressources disponibles pour les applications cibles ainsi que leur variations. Chaque agent utilise des informations statiques et dynamiques sur les applications dont il est responsable et sur l'environnement auquel il a accès pour effectuer ses choix. Il peut alors utiliser les systèmes chargés de la gestion de l'environnement pour l'alllocation des fichiers et des calculs aux différentes ressources.
La prochaine génération d'agents AppLeS devrait être des modèles, des agents "types", dédiés à des classes d'applications structurellement équivalentes. Dans ce document, nous introduisons un modèle d'agent dédié à l'ordonnancement des applications de type Parameter-Sweep (des applications constituées d'un grand nombre de tâches de calcul indépendantes pouvant partager des fichiers d'entrée). Nous avons conçu un algorithme d'ordonnancement général capable de tenir compte des modifications de structure ou de performances de l'environnement et d'utiliser un certain nombre d'heuristiques pour l'allocation des données et des calculs aux différentes ressources. Nous avons développé un simulateur avec lequel l'agent peut interagir pour pouvoir évaluer et comparer ces heuristiques. Ce simulateur nous a permit de réaliser rapidement un grand nombre d'expériences dans un environnement paramétrable à volonté. Notre point de départ à été l'utilisation d'heuristiques tirées de la littérature et de les adapter à notre type d'application. Ce document présente la mise en œuvre du simulateur et explique comment il va être utilisé pour obtenir de nouveaux résultats concernant l'ordonnancement dans des environnements instables de type *grille*.

Mots-clés: Ordonnancement, Méta-computing, Calcul distribué, Grilles, Applications indépendantes, Partage de fichiers

1 Introduction

The *Grid* [20] is one of the most exciting emerging technologies in the area of distributed computing and numerous projects focus on the design and implementation of different parts of the Grid vision [19, 22, 12, 38, 5, 29]. Those projects aim at providing operating systems, environment, and tools to harness the tremendous amount of resources that are expected to be available for production runs of large scale applications. Those resources include computing and storage devices interconnected by various networks. The Grid is aimed at inherently dynamic environments: resources are **shared** so that the contention created by multiple applications often creates fluctuating delays and qualities of service, and resources are **federated** which means they are transient and with difficult to predict availability as there can be no centralized control. Furthermore, resources are **heterogeneous** and may not perform similarly for different applications. This dynamic and heterogeneous aspect of this environment makes it difficult to schedule applications in a way that maximizes appropriate (application-level or system-level) performance metrics. There is a definite gap between theoretical works concerning scheduling and what can be actually implemented effectively in real distributed systems. The main goal of this research work is to explore ways in which this gap can be bridged by designing practical scheduling algorithms and by adapting well-known scheduling heuristics in order to accommodate Grid-like environments A basic requirement is that these algorithms should be flexible enough that different strategies/heuristics can be easily "plugged in" to account for diverse applications and environments. The evaluation and validation of these heuristics and strategies as well as the impact of Grid characteristics is accomplished by simulations and experiments. As a starting point we focus on application-level scheduling for a restricted class of applications: *parameter-sweeps*; but will consider other applications in future developments (e.g. applications structured as more general DAGs).

We define a parameter-sweep application as one that is composed of a *large* number of independent tasks. By *large* we mean that the number of tasks is usually one order of magnitude larger than the number of available computing resources. We model the computing environment as a federation of clusters containing hosts that can be used for computation. The performance of the hosts and network links are dynamic and each cluster may or may not have a storage device accessible to all its hosts. From now on, we will call this specific environment the *grid* (for the sake of brevity) as opposed to the *Grid* which we view as the software infrastructures and methodologies described in [20].

The problem is to schedule a complete run of the application onto the grid in a way that optimizes a chosen performance metric. For now we address one of the most intuitively appealing metrics: the *makespan* [30] of the application, that is the time elapsed between the beginning of the application and the completion of its last task. Other metrics will be considered in future work (e.g. various *cost* models).

Various scheduling strategies and heuristics can be used in a view to reducing the makespan. However, it is very difficult to conduct experiments that would allow comparisons in different grids in a reproducible manner. A good way of tackling this problem is to use a simulator. We have therefore implemented a simulator as part of our software (see Section 7). This simulator allows us to simulate very diverse types of applications and grids while obtaining perfor-

mance results quickly and easily. Furthermore, since the simulator is integrated with the rest of our software, it is possible to design, test, and evaluate scheduling algorithms in "simulation mode". Once a scheduling algorithm is deemed appropriate it can then be taken out of simulation mode with absolutely no change to the algorithm's implementation.

Section 2 describes related work. Sections 3 and 4 describe precise models for the the application and the grid. Section 5 describes the skeleton of our scheduling algorithm and different heuristics. Section 6 describes the template's implementation whereas Section 7 focuses on the simulator. Finally, Section 8 presents an early simulation experiment and discusses how such experiments will be used to obtain general results concerning Grid scheduling.

2 Related Work and Contribution

The numerous works related to our effort can be roughly categorized as either *applied* or *theoretical*. As stated in the introduction, our work is located somewhat in between since (i) we design our scheduler with the goal to being implemented as part of deployed Grid applications and (ii) we set to exploit results provided by theoretical scheduling works and try to adapt them to Grid environments.

As stated in the introduction, we eventually plan to use the Grid as a software platform for deploying our scheduling strategies. We must therefore use and interact with projects that aim at providing Grid infrastructures [19, 22], as well as Grid middlewares and services [38, 12, 5, 29, 28, 43]. Section 6.3 describes our first steps towards an implementation of our software using some of the software generated by those projects. Many references to projects that are mainly focused on scheduling issues can be found in [7]. This research is part of one of these projects, AppLeS [9, 8, 10], which is focused on developing a methodology supported by the dynamic forecasting services of the Network Weather Service (NWS) [43] for application scheduling in shared computational Grid environments. AppLeS methodology, supported by NWS forecasting services, has yielded a set of high-performance, dynamically schedulable Grid applications, and provided compelling evidence that dynamic scheduling is key to achieving Grid application performance. However, the process of building AppLeS/NWS-enhanced applications is often time- and labor-intensive as it may require considerable customization of the client application so that it can receive directives from the scheduler and be deployed on the Grid. This often requires a tight application-scheduler integration that makes the implementation difficult to maintain and upgrade. Accordingly, a new trend in the AppLeS project is to focus on ways to provide modular and extensible implementations of AppLeS/NWS-enabled applications. The idea is to consider *classes* of structurally similar Grid applications and develop *templates* that are able to capture the relevant performance behavior of applications having similar structures. A template can be used as a framework for developing distributed applications whose instantiations will result in a Grid application which knows how to dynamically schedule itself. The overall goal is to provide a usable, performance-oriented, rapid application development environment for the user. This work is a first attempt at designing and implementing a template: the *Parameter-Sweep Template* (PST) that targets scheduling of applications composed of indepen-

dent tasks in a Grid environment (see Section 6).

The Nimrod project [3, 4] (and Clustor, its commercial counterpart available from [2]) is targeted to computational applications based on the "exploration of a range of parameterized scenarios" which are similar to the applications we are targeting. A current effort, Nimrod/G, aims at providing an implementation of Nimrod on top of Globus [19]. Our work differs from the Nimrod project in multiple ways. First, as far as implementation and deployment are concerned, we do not make any restriction on the Grid software being used. We show in Section 6 that our software is built so that it can use any available software as long as that software complies with a few requirements and provides a few key functionalities. We expect our work to be deployed on top of Globus as well as other software platforms. A large part of Nimrod is concerned with user interfaces whereas we keep this part of our project to a minimum (but general enough that appropriate interfaces could be built if needed) so that we can focus primarily on scheduling research. To the best of our knowledge, the scheduling algorithm in Nimrod does not try to monitor Grid conditions as we do with the NWS. In addition, a contribution of our work is to integrate the file locality aspect with the scheduling. We believe that many parameter sweep applications will require large amounts of data to be transfered to remote sites and that careful scheduling of these transfers can have a major impact on the application's performance. Our work should in fact be applicable to the Nimrod framework and one can realistically imagine an implementation of Nimrod that uses our framework to do data-aware scheduling on the Grid.

Another effort for scheduling independent tasks on the Grid is described in [13] and is part of the NetSolve [12] project. That work, called *task farming*, does not at the moment exploit information such as data sharing among tasks. However, there is an very active collaboration between the AppLeS, NetSolve, and IBP teams to enable the exploitation of such data patterns. The research in this work will provide scheduling mechanisms that will be used in the infrastructure provided by NetSolve/IBP. Our current implementation makes use of NetSolve as a possible interface to the Grid, and we are currently investigating using IBP for data storage.

A large number of theoretical works attack the question of mapping a set of tasks onto a heterogeneous set of processors [26, 15, 27, 36, 42]. These works usually propose models that are supported by simulation results with suitable sets of assumptions. A large subset of that body of work addresses the specific case where the tasks are independent which interests us in the first step of our research. Among these are works that take the *batch scheduling* approach [23, 24]: tasks can be grouped in batches and a fixed overhead is paid for sending a batch to a resource. In that setting, large batches lead to low overhead but may cause load-imbalance whereas small batches lead to high overhead but achieve good load-balance. We decided not to use such a model for our computational environment for two reasons. First, the concept of a fixed overhead for batches of tasks is difficult to justify in a Grid environment as connectivity to difference resources is disparate. Second, even though the batch model is applicable to some subset of the applications we are considering it does not seem generally applicable in our setting (e.g. the fixed overhead assumption is violated). This is mostly due to the fact that our model pays special attention to data storage issues, in a view to being more realistic.

A recent reference concerning the scheduling of independent tasks without

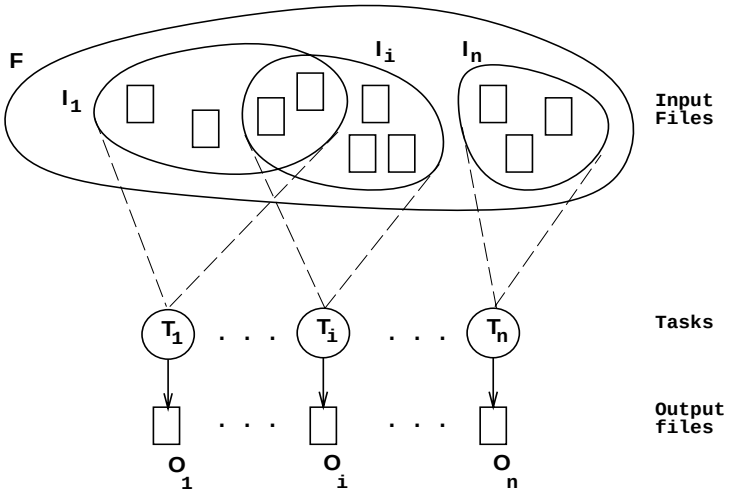


Figure 1: Application model

batch scheduling is [31] and it is a good source of commonly used heuristics. That paper considers a scheduler that reacts to incoming requests on-the-fly or that processes groups of requests at a time. It is the second mode of execution that can be applied to our work since in our current model we initially know all the tasks that are to be performed. The model in [31], unlike ours, does not take into account data storage and data sharing issues. We propose a strategy for deciding on the intervals between our scheduling events (see Section 5.6.1) aiming at accounting for the Grid instability whereas the scheduling events in [31] are based on task inter-arrival times. We describe optimization techniques that are relevant in a view to an actual implementation of the scheduler on the Grid and also propose new heuristics that are adapted to a Grid environment. The simulation results in [31] assume truncated Gaussian distribution for task execution times whereas we designed our simulator to perform simulations with actual traces from real measurements (e.g. from the NWS). Finally, we allow dynamically changing number of resources (accounting for resource failures or additions).

Finally, a number of recent references address the question of simulating heterogeneous distributed environment for the purpose of evaluating scheduling strategies. The motivation for simulation is that a large number of performance data can be obtained very quickly and with 100% reproducibility. However, only a few projects provide actual implementations. Among such projects one can mention Bricks [41] and Osculant [1]. Bricks seems to be more general as it is designed to be extensible and can integrate various components in the simulation. The simulator that we have developed for our purpose (see Section 7.1) does not aim at providing a general Grid simulator and is really tailored to our application framework (parameter-sweep for now). Futhermore our approach is event-driven whereas Bricks uses queuing theory. It could be interesting to see how a framework like Bricks, when it becomes available, could be used for our purposes.

3 Application Model

We define a parameter sweep application as a set of n independent computational tasks $\{T_i\}_{i=1,..,n}$. By *independent* we mean that there is 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 other input is *negligible* in terms of byte size. This assumption is motivated by the applications that we are currently targeting (see Section 6.1) and the term *file* should be understood as *datum*. Let F be the set of all files that are input to at least one of the tasks. For each task T_i we denote the set of its input files by $I_i \in 2^F$. From this definition it is clear that an input file may or may not be shared by multiple tasks. Each task produces an output (which may consist of one or multiple files, or may be of *negligible* size). With no loss of generality we denote the output of task T_i by O_i and view it as a single file (of possible null size). At the moment, we assume that a task runs on a single host (no intra-task parallelism). Figure 1 depicts our application model and shows an example of input file sharing among tasks.

4 Grid Model

We view the grid available to the user as a set of clusters $\{C_j\}_{j=1,..,k}$ that are accessible via k network links denoted $\{L_j\}_{j=1,..,k}$. Cluster j contains m_j hosts that can be used for computation; we denote these hosts $\{H_{j,k}\}_{k=1,..,m_j}$. A host can be any computing platform, from a single-processor workstation to an MPP systems, and can be made available in interactive or batch mode. The different model for interactive or batch mode should be encapsulated in our *running time estimates* (see Section 5.2) and is not part of our model per-se. We allow for a dynamic number of available hosts within a cluster as well as a dynamic number of clusters. In addition we assume that a shared storage facility is available at each cluster so that files can be shared among the processes running on different hosts in the cluster. From now own we will call hosts and network links *resources* and all are conceptually modeled as FIFOs. Figure 2 depicts this model. Let us justify and discuss our assumptions.

From now own, we will call the user's host the *origin*. The assumption that there is exactly one link between the origin and each cluster is not entirely realistic. However, it is generally the case that the complexity of real network infrastructures precludes the use of tractable models. For instance, the possible dynamic routing of messages in an Internet setting cannot be captured by our model. We do not have immediate plans to move towards more complex models of the network and claim that our model suffices for our purpose.

The storage infrastructure that we require at the cluster level can be implemented in various ways including distributed file systems (NFS, AFS, etc.) or lower-level facilities [18, 6]. Production clusters usually provide at least one of these facilities. For the time being we assume that storage space is infinite or in other words that it is always possible to store any file within any cluster. This may not be realistic in real environments for certain scenarios (multiple applications sharing the storage, several extremely large data file, etc.). However, taking into account limited storage would require the investigation of different policies (fixed quota, replacement policy, etc.) and we leave this for future work. However, we give elements for building heuristics that we expect be more

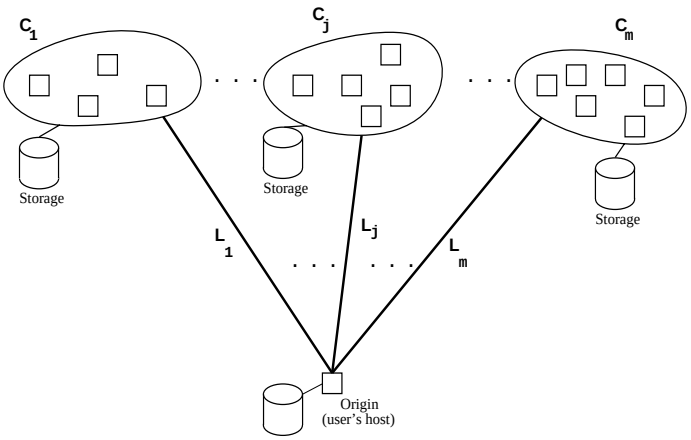


Figure 2: Grid model

applicable in a limited storage setting.

One can notice on Figure 2 that we do not model network links between clusters. However, in a real environment, hosts on different clusters could exchange files directly rather than obtaining them from their original location (e.g. on storage attached to the origin). We leave this kind of possibilities for future work. Similarly, we do not model network connections within a cluster. This implies that we cannot model contention when multiple hosts in the same cluster try to access the shared storage. The expectation is that since we are primarily targetting wide-area networks, the bottleneck of network transfers should be the links between the origin and the clusters. Modeling intra-cluster contention is left for future models.

At this point we do not impose any constraints on the performance characteristics (both static and dynamic) of the difference resources in the grid (networks and hosts). We just require that there be a way to obtain some estimates of the current achievable performance on each resource (e.g. including possible queue waiting times and actual computation times). Our scheduling decisions are based on those estimates. We do not impose any constraints on the accuracy of these estimates. However, if guarantees on the estimates' accuracies are available our scheduler should be able to make use of them. See Section 5.2 for a discussion of the estimation procedures. We believe that the simplifying assumptions in our grid model make it possible to obtain initial meaningful results while keeping the simulation reasonably simple.

5 Scheduling

5.1 Input/Output Policies

In the description of our application model in Section 1, we did not specify anything about the possible/desirable *locations* of the input and output files to the different tasks. As far as input files are concerned, there are multiple scenarios. For now we will assume that all the input files, of known sizes, are initially available to the origin; and according to our current assumption of no

inter-cluster communication (Section 4), the origin is then the only site that is authorized to distribute input files. Future scenarios will include cases where the input files are already distributed and possibly duplicated at different locations (e.g. Web servers over the Internet) before a run of the application.

We currently assume that the output files generated by the tasks must be returned to the origin. Other scenarios include ones where the output files must be re-directed to other locations (e.g. for post-processing or storage in databases) and are left for future work. Depending on the application, the size of the output generated by a task may not be known a-priori. We consider cases where the size of each output files is known, or cases where the sizes must be estimated (see Section 5.2).

5.2 Resource Performance Estimation

5.2.1 Running Time Estimates

The design of most scheduling algorithms is based on the fact that *some* estimate of the running time of a task on a resource is available. This is especially true when the tasks to schedule exhibit dependencies. In the case of independent tasks, there are schemes such as simple work-queue algorithms (e.g. self-scheduling [23]) or work-stealing strategies [11, 37] that may not need any a-priori knowledge about the expected running time of the tasks to be scheduled. Since the target applications for this work are composed of independent tasks, one may wonder why we pay attention to performance estimation techniques. The first reason for taking into account running time estimates is that the set of tasks and available hosts may exhibit *affinities* (as defined in [31]). This means that different resources are best for different tasks. It is often desirable to assign a task on the machine that performs it the fastest, that is the one with the smallest estimated running time for that task. To keep our work general and applicable to a wide variety of applications we then need to exploit possible affinities. The second reason is that, in a view to producing a more realistic model, we consider network links as resources and subdivide each task of the application into dependent subtasks (Section 5.4 describes this in detail). The efficient scheduling of dependent tasks must use estimates for task running times. A third reason for needing running time estimates is that the performance of the application is generally improved thanks to scheduling heuristics that make use of those estimates (see Section 5.6.3). In order to further verify the need for these estimates, we perform simulations to compare our scheduling algorithm to the standard self-scheduled work-queue.

The running times that may need to be estimated are the execution times of the computational sub-task of each T_i on each processor $H_{j,k}$ and the file transfer time for each input and output file over each network link. Computations of these estimates can make use of three different types of information: (i) user-supplied; (ii) historical; (iii) forecasted. Let us discuss each type in more detail. The first type is pretty obvious. It is highly likely that the user possesses some knowledge about his/her application (e.g. expected execution time for single tasks or comparisons between tasks). We must allow our estimation techniques to take any input from the user into account. Second, historical information is usually application-related or grid-related time-stamped data series that can be stored and retrieved in some database. Examples include past load averages

on processors, past network bandwidth, past tasks execution times, past file transfer times, past queue waiting times (for hosts that are available via a batch system). Usually, grid-related historical information can be retrieved from deployed Grid information services like the NWS [43], the MDS [17], or directly from Grid resources, whereas application-related historical information may be captured by the scheduler itself as the application is running. Third, forecast information can be obtained directly from forecasting services such as the ones provided by the NWS or computed by our scheduler based on historical information, or based on a mixture of historical information and NWS forecast. In fact, it is even possible to use the NWS forecasting modules to perform forecasting on arbitrary time series, including the application-related historical information gathered by our scheduler.

As seen in Section 5, our scheduling algorithm would benefit from estimates of expected completion times of currently running subtasks. This is a little different from estimating just running time because more information is available. Indeed, for each currently running sub-task, the scheduler knows exactly when the sub-task was started and what was its predicted running time at that time. If reasonably accurate, that information should provide good insight when trying to estimate the remaining time to completion. In some sense, this is similar to trying to obtain an estimate of the percentage of the sub-task that has already been executed. Such an estimate may be provided by the application itself (this is however not the case for our initial target applications described Section 6.1) or computed with simple heuristics or more sophisticated techniques using historical information. A simple heuristic could be as follows. If the sub-task currently running is not overdue with respect to its initial running time estimate, then one assumes that it is going to complete according to that estimate. If instead it is overdue, then one assumes that it is going to complete in some constant amount of time. Such a heuristic may actually prove effective for environments that have fairly stable and predictable behaviors. Other heuristics and more sophisticated techniques can make use of historical grid-related information to try to actually understand why the task is overdue and how badly it has been slowed down. At this time, such techniques are under investigation. Our current approach is to use past CPU load measurements from the NWS in order to see what portion of the CPU was dedicated to the sub-task under consideration and hence estimate what progress has been made.

Other kinds of estimates might be needed in order to help our scheduler making better choices. For instance, if the expected size of the output file of a task is not known a-priori, it may be possible to estimate it based on previous runs, assuming that tasks will exhibit similar patterns. For instance, one can assume that the volume of output generated by all tasks are roughly the same, or one can try to establish a relationship between the volume of input and the volume of output (e.g. linear regression [16]). This issue is not critical at the moment as the users of our initial target applications know precisely the amount of output to expect. Nevertheless, we will conduct some experiments in our simulation in order to assess the impact of unknown output sizes on the execution.

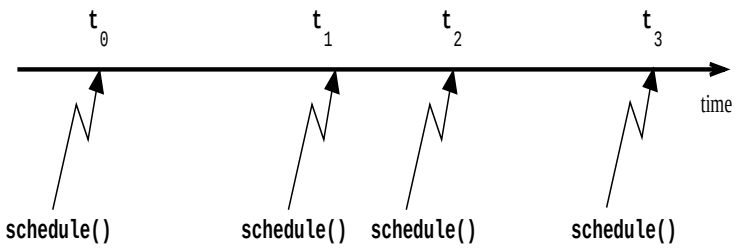


Figure 3: Scheduling scheme

5.2.2 Quality of Information

We have already stated that we impose no constraints on the accuracies of the running time estimates, or in other words the *quality* of the information concerning performance prediction. Intuitively, stable resources will lead to good quality of information whereas instable resources will make it difficult to obtain reasonably accurate estimates. The question is to know how the quality of information impacts scheduling algorithms. And furthermore, how should schedulers adapt to the presence of low quality information (noisy or chaotic performance measurements) ? The answer to those questions depends heavily on the resource behaviors and on the heuristics used within the scheduling algorithms. We use simulations to examine multiple ways of obtaining estimates in order to make comparisons and understand the impact of their accuracy on the schedule. We consider cases where all estimates can be computed (i) with one-hundred percent accuracy; (ii) with a bounded percentage error; (iii) with techniques that aim at discovering the nature of the probability distributions of performance characteristics; (iii) with forecasting techniques that use historical data (our goal here is not to design new forecasting techniques and we just use NWS standard predictors combined with our application-related historical information). Section 7.1 describes the different models that can be used for resource performance. Simulations combining models for resource performance with the aforementioned models for the computation of estimates should give us good general ideas of the impact of quality of information on scheduling and we present results in Section 7. We believe that this area is of great relevance to Grid scheduling and its impact on parameter-sweep application is only our first exploration of that domain.

5.3 General Scheduling Scheme

Figure 3 shows the general scheduling scheme. Let us assume that we have designed a scheduling algorithm (let us call it `do_plan()`) that takes into account user-supplied, historical, and forecasted information about the grid and the application, and generates a *plan* (a series of actions) for the scheduling of tasks that have not yet been assigned to hosts. In our current implementation, the scheduling algorithm runs on the origin host. In a perfectly predictable and stable computing environment, `do_plan()`should be called only once at the beginning of the application. However, in a Grid environment, scheduling decisions lose their validity as the environment changes and the schedule of the application must be re-evaluated at times. We call these times *scheduling*

events and denote them $\{t_i\}_{i=0,..}$ as shown on the figure. The frequency of the scheduling events must be high enough to account for the level of instability of the grid.

It is a common assumption that an execution of a scheduling algorithm is instantaneous (e.g. in [31]). In that case, one can be conservative and call it more often than really needed (a straightforward possibility is then to have frequent, evenly spaced scheduling events). It is also a very convenient assumption as it implies that the environment and the computations do not evolve or make any progress while the scheduling algorithm is running. This assumption is usually justified by the fact that scheduling algorithms often implement heuristics that have polynomial time complexity as they try to approximate solutions to NP-complete problems [25] and that polynomial time if "good". Even so, when the number of tasks and resources is large, it can become impractical in a real implementation to call a polynomial-time algorithm too often. More importantly, our scheduling algorithm may need to dynamically obtain information about the current and past state of the grid from some information service, and that may cause some overhead. Finally, the computation of estimates for the completion times for currently running sub-tasks (see Section 5.2) may cause overhead if it relies on historical data rather than simple heuristics. In fact, for large pools of resources, it will probably not be possible to gather all relevant information at each call of `do_plan()` and the scheduler will have to use out-of-date information. There is then an interesting trade-off to be made in between the amount of out-of-date information and the execution time of the scheduling algorithm [32]. It may be that a scheduling algorithm that takes the time to obtain better information could outperform one that uses "too much" out-of-date information. Lastly, even though the scheduling heuristics we are considering in this work are not prohibitively expensive (complexity-wise), one can imagine that more sophisticated ones may lead to longer execution times for the scheduling algorithm. Our simulator allows us to simulate different running times for the scheduling algorithm (i.e. meaningful at the simulation time-scale).

To address the aforementioned issues and determine how far apart the scheduling events should be, we investigate simple techniques that aim at: (i) tuning the intervals in between scheduling events in order to adapt to the instability of the grid without being overly conservative; and (ii) taking into account that the application and the environment have time to evolve while the scheduling algorithm is trying to make decisions. Section 5.6.1 gives the details of these techniques and also explains why more scheduling events may be generated "on-the-fly" to account for dynamic additions of resources to the grid.

5.4 Task Model

Our scheduler assigns a task T_i onto some host $H_{j,k}$. For the task to execute on that host, all the input files in I_i must be present in the shared storage at cluster C_j. Some of these files may already be there (from previous file transfers for other tasks), and some may need to be transfered from the origin. Then the actual computation can take place and its output must be returned to the origin (see Section 5.1). The execution of a task T_i on $H_{j,k}$ can be viewed at the execution of a simple dependency graph of sub-tasks, as depicted in Figure 4. The sub-tasks for input and output file transfers must be scheduled on a network link to some cluster and the central computational task must be scheduled on

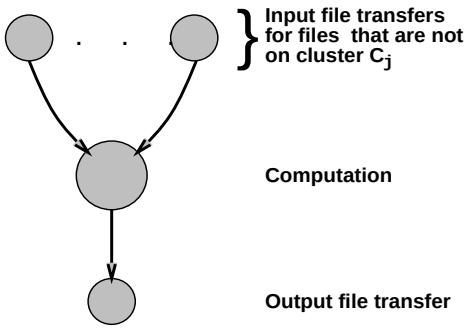


Figure 4: Task T_i on host $H_{j,k}$

```
do_plan(){
    (1) compute t_{i+1}
    (2) create a new Gantt Chart, G
    (3) foreach currently running sub-task
                compute an estimate of the completion time
                fill in the corresponding slots in G
    (4) select a subset of the tasks that have not started execution: T
    (5) compute Δ
    (6) until each host has been assigned work after t_{i+1} + Δ
            use heuristics to assign slots in G for tasks in T
    (7) convert G into lists of instructions
}
```

Figure 5: Scheduling algorithm

a processor in that cluster. In the current implementation of the PST software and of its interaction with Grid components the splitting of each task of the application into a dependency graph of sub-tasks can be easily accomodated, and we anticipate this to be the case for future implementations as well.

5.5 The Scheduling Algorithm

We assume that at each scheduling event our scheduler has access to: (i) the current topology of the grid (number of clusters, number of hosts in those clusters, network links); (ii) the number and locations of copies of input files; (iii) the list of sub-tasks currently running. This is natural from an implementation standpoint.

Figure 5 shows the general skeleton of the scheduling algorithm. Let us go through all the steps involved. Assuming that the call to `do_plan()`takes place at the i^{th} scheduling event, step (1) of the algorithm determines when the next scheduling event should occur (by computing t_{i+1}). Possible ways of performing that computation are presented and discussed in Section 5.6.1. Step

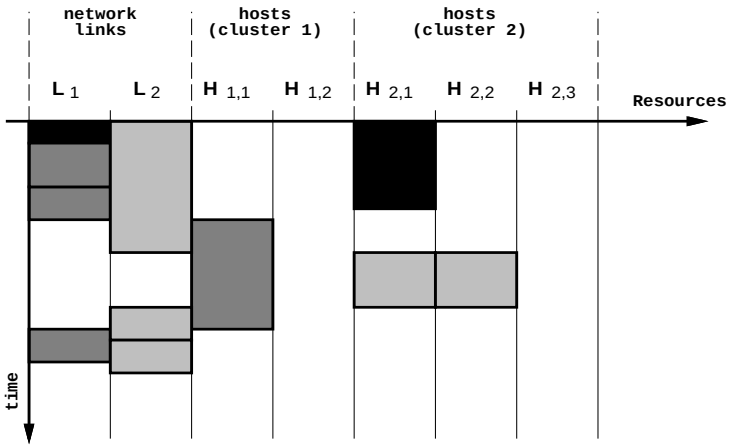


Figure 6: Sample Gantt chart

(2) consists in creating a Gantt chart [14] with as many columns as resources (hosts and network links). Let us note that in the case of a multiprocessor computing resource, we would have to create as many columns in the Gantt chart as processors available on that host. The Gantt chart will be used in the rest of the algorithm to store task-processor assignments. Figure 6 shows a sample Gantt chart associated with a grid composed of two clusters with two and three hosts in each cluster respectively (the other elements in the figure will be explained later).

Step (3) takes care of the tasks that are currently running. In the current version of our scheduling algorithm we do not allow task migration. This means that once a sub-task has been started on a resource it will complete on that resource unless the resource fails. The Gantt chart must be seen as an drawing board for the future of the computation and as such, one must first draw elements that can not be changed, such as the sub-tasks already running. Hence, for each running sub-task one must estimate when it is going to complete according to methods that were introduced in Section 5.2 and insert it in the Gantt chart. No subsequent sub-task assignment will occupy the slots occupied by currently running sub-tasks. Note that we view the processor load generated by processes that are not part of our application as *background* load, that is as a processor slowdown over which we have no control. The running time estimates aim at taking into account that load for completion time predictions.

Two examples of running sub-tasks are shown on Figure 6 as black-filled rectangles in slots at the beginning of the chart. Once step (3) has been performed, it becomes possible to plan assignments of new sub-tasks to resources. The number of resources in the Gantt chart and the number of unscheduled tasks may be both fairly large and may hence make the scheduling algorithm too expensive if it were to consider them all. Steps (4) aims at bringing a solution to this problem when it arises by allowing only a subset of the unscheduled tasks to be considered. The choice of this subsets, say T, depends on the complexity of the decision making process performed in step (6) and on the number of available hosts. Note that even though there is no task ordering in our applications, the choice of T should take into account possible input file sharing

between tasks. Heuristics for making that choice are discussed in Section 5.6.2.

The rest of the scheduling algorithm considers the tasks in T and assign slots to their sub-tasks in the columns of the Gantt chart for hosts and corresponding network links. The chart gets then filled along the time axis as the scheduling algorithm makes progress. Sample assignments are shown on Figure 6. Since the scheduling algorithm will be called again at t_{i+1}, it is not necessary to assign slots in the Gantt chart for sub-tasks that would start after t_{i+1}. This is important if the heuristics used to assign sub-tasks to resources have some non-negligible computational cost. However, there is some uncertainty on the execution time of the sub-tasks. In particular, if sub-tasks execute faster than estimated some host may remain idle until t_{i+1}. One possible way to account for this uncertainty is to fill in the Gantt chart until each resource under consideration has been assigned a sub-task that is estimated to start after time $t_{i+1} + \Delta$ where Δ is to be chosen heuristically. The higher the confidence in the estimates obtained for the running times while filling in the Gantt chart, the smaller can Δ be chosen. However, being conservative in choosing the value of Δ should not have a dramatic effect on the running time of the scheduling algorithm (after all, the heuristics in use had better be in polynomial time). And choosing too small a value can lead to idle time. For instance, choosing $\Delta = t_{i+1} - t_i$ might be an acceptable choice based on the assumption that running-times estimates are at most twice as large as actual running-times. Another possibility is to dynamically tune the value of Δ, starting the scheduling algorithm with a large value and modifying it at each scheduling event to account for fluctuations in grid stability. Our simulations will explore a few of those strategies.

Step (6) is the core of the scheduling algorithm as it decides on assignments of sub-tasks to resources. Until the Gantt chart is *full*, meaning that no sub-task has been assigned to start after time $t_{i+1} + \Delta$, one uses heuristics to make decisions for assigning sub-tasks of tasks in T to slots in the chart. A possible heuristic would be to pick a task at random among the ones in T and assign its computation sub-task to the host that leads to the smallest execution time of that sub-task. Different heuristics are detailed in Section 5.6.3 and we compare them all in our simulations. Examples of slot assignments are depicted on Figure 6 as gray areas. The slots assigned to the sub-task of one of the T_i are shown in dark gray. Two input files are transfered on link L_1, followed by the computation on host $H_{1,1}$, and the output is transfered back to the origin via link L_1 after the computation is completed. In light gray are shown two tasks that both take the same file as input. That file is transfered on link L_2 and both tasks can then start simultaneously on hosts $H_{2,1}$ and $H_{2,2}$. Each tasks produces one output file and these files are both transfered back to the origin via link L_2.

5.6 Strategies and Heuristics

Our discussion of the scheduling algorithm in the previous section is focused on the general scheduling scheme and structure of the algorithm. All the "smarts" of the scheduler reside in the heuristics employed to make decisions. The following sections describe several heuristics and techniques for different parts of the algorithm, some from the literature, and some that we specially designed to answer our needs. We expect that the bulk of the research fostered by this work will take place in the design of new heuristics. The main rule of thumb in

designing heuristics is that they should not be overly sophisticated because we want them not only to be computationally inexpensive but also because given the dynamic nature of Grid environments it seems intuitive that little could be gain by increasing sophistication beyond a certain point.

5.6.1 Scheduling Events

How is t_{i+1} to be computed in step (1) of the algorithm ? One possibility is to set $t_{i+1} - t_i$ to a constant value for all i. That value should be larger than the average running time of a task. However, in a dynamic, heterogeneous grid environment there might be no clear estimate of that average. In fact, a dynamically tuned value for $t_{i+1} - t_i$ seems more appropriate. At each call of `do_plan()`, one can compute some measure of the *deviation* with the planned schedule. A way to compute this deviation is to add a step (8) in the algorithm where one computes which sub-tasks should be running on which resources at time t_{i+1}. When `do_plan()`is called at time t_{i+1}, one then compares the currently running sub-tasks with what was predicted. The deviation can then be defined as a percentage of correct predictions for the sub-tasks that should be running. This is not a perfect solution as it does not take into account individual resource behaviors. For instance, if half the resources are very stable and the other half are very instable or it is likely that the deviation will always lie around 50%. A possibility then is to assign weights to individual deviations to take into account that no matter how small $t_{i+1} - t_i$, some resources will be unstable. The goal is to detect unstable resources and lower their weight in the computation of the global deviation with the planned schedule. Based on the measure of deviation, $t_{i+1} - t_i$ can be increased or decreased. A common strategy is to allow it to increase slowly as long as the deviation is below some threshold, and decrease it *fast* when that threshold is overcome. We wil use variations on this theme in our simulations.

To summarize, here are the three strategies we simulate for choosing the t_i sequence:

(S1) constant value for $t_{i+1} - t_i$,

(S2) $t_{i+1} - t_i$ based on *deviation*,

(S3) $t_{i+1} - t_i$ based on *deviation* with resource weighting.

For all three strategies we need to chose the initial value for $t_1 - t_0$. Choosing a conservative value for strategy (S2) and (S3) is a good choice since it will be allowed to increase as the execution makes progress. A possibility is to determine the cost of the scheduling algorithm and start with the maximum allowed calling frequency on the origin (e.g. with a user-specified percentage of the CPU allowed for scheduling). For strategy (S1) we will use a number of values in our simulations.

The strategies described above do not explicitly take into account the fact that the number of resources is dynamic. The important case is when new resources become available in between scheduling events. At the moment, these resources would stay idle until the next scheduling event. A way to prevent this is to call `do_plan()`each time new resources are added. This is a practical solution because it is easy to implement and because we do not expect to

have resources added too often. When resources become unavailable, it seems reasonable to wait for the next scheduling event rather than re-scheduling. We simulate environments with varying resource pools and compare strategies that never re-schedule, re-schedule for resource creations, for resource deletions, or for both.

As we mentioned in Section 5.3, we do not assume that running the scheduling algorithm is instantaneous but that on the contrary the computation may make progress during a call to `do_plan()`. The difficulty without that assumption comes from the fact that we have no accuracy guarantee on the estimates for the completion time of currently running sub-tasks (see Section 5.2). The idea is then simply to leave "a few" of the previously scheduled tasks untouched, so that if a resource becomes available while `do_plan()`is running it can start on previously scheduled tasks. This technique may prove useful to avoid idle time in the case of a scheduling algorithm that runs for a non negligible amount of time (see Section 5.3). We perform simulations to evaluate the validity of such a technique. From the implementation point of view, one just needs to have the scheduling algorithm check on sub-task completions regularly. This is not an issue given the architecture of our software (see Section 6) and can be implemented without difficulties.

The discussion in this section may not be relevant for some versions of the scheduling algorithm that demand very little computation and we take this into account in our simulations.

5.6.2 Reducing the Task-space

Steps (4) of the scheduling algorithm allows reductions of the number of tasks considered by the algorithm. Parameter-sweep applications contain very large number of tasks, and an application that consists of tens of thousands of tasks will not be unrealistic in the close future. When confronted with such large numbers of tasks, some scheduling heuristics may lead to prohibitive execution times (of the scheduling algorithm). For instance, a common behavior for step (6) of the algorithm would be to inspect the whole task space for each task/host assignment. Even though it is possible to limit the number of assignments to be performed (by lowering the Δ value as explained in Section 5.5), extremely large task spaces could be a problem. There are several possibilities for limiting the number of tasks. One can chose T as a random subset of fixed size of the set of all tasks. This may cause problems because exploitable patterns in the task space may be ignored. Another possibility is to select subsets with a maximal number of tasks sharing at least one *large* input file in a view to maximizing data re-use. Such a heuristic will probably be effective in limited-storage environments (left as future work for now). The size of T depends on the complexity of step (6) in the algorithm as well as on the number of hosts. We perform a few simulations to understand the impact of task-space reduction on the scheduling algorithm running time and the validity of the schedule.

One could think of also limiting the number of hosts in order to cut down on the computation time for `do_plan()`. This is not as easy to justify. Indeed, this would leave hosts idle which in general is not desirable for the application (the trade-off is in between running an expensive scheduling algorithm or not using some of the resources available to the application). Besides, from a practical standpoint, we do not expect our implementation to use prohibitively large

numbers of hosts (or at least in the close future).

5.6.3 Task/Host Selection Heuristic

In this section we describe the main heuristics for building a schedule for the application, the ones that can be used in step (6) of the scheduling algorithm. All the heuristics that we consider aim at iteratively choosing task/host assignments. Formal description of all the heuristics mentioned hereafter are given in Appendix C.

Three common heuristics
Three heuristics for completely independent tasks (no sub-tasks, no input file sharing) are proposed in [31]. They are based on work in [25] (in which the scheduling problem is recognized as NP-complete). These heuristics are called respectively *Min-min*, *Max-min*, and *Sufferage*. We refer to [31] for their complete description and we just describe the main ideas.

The **Min-min** heuristics considers all the tasks in T and for each task computes its completion time on each available host. It is then possible to compute for each task its Minimum Completion Time (MCT) and determine the host that achieves it. The task that has the smallest MCT is then assigned to the corresponding host. That task is removed from T and the heuristic iterates until enough tasks have been assigned resources. It is said in [31] that the complexity of this heuristic is linear in the number of resources and quadratic in the number of tasks. The rationale behind Min-min is that assigning tasks to hosts that complete them fastest will lead to a overall reduced makespan.

The **Max-min** heuristic is very similar to Min-min and has the same complexity. The only difference is that it assigns the task with the largest MCT to the host that achieves it. The rationale behind Max-min is to allow *long* tasks to run concurrently with *short* tasks. This is not really an issue if the tasks in the application are all identical in terms of computational cost and amount of I/O.

The idea behind the **Sufferage** strategy is that a host should be assigned to the task that would suffer the most in terms of completion time if not assigned to that host. The heuristic, as described in [31], computes the completion times for all tasks in T on each host assigns tasks to the host leading to MCT. However, that assignment may be canceled for the benefit of a task with higher sufferage value (difference between best and second-best assignment in terms of completion time). Once a task has been assigned that way, it is considered assigned to the host for good. Our version of Sufferage (detailed in Appendix C) is actually a little more involved as it computes global maxima of sufferage values, as opposed to the cancellation/re-assignment described in [31].

Extending sufferage
Our intuition is that a sufferage strategy should lead to better performance than pure Min-min or Max-min. In our setting, we expect the fact that input files may or may not be present on a remote cluster to have a great impact on the sufferage value. In other terms, a task may be running faster than some other on some host, but the priority should be given to the task whose input files are more available to that host if the network is slow or if the files are large. Trying to fully explore the possibilities here will undoubtedly lead to an exponential

time algorithm, and we believe that a simple metric like the sufferage value leads to good approximations of an optimal schedule. As it will be seen in the simulation results, applying the classical version of Sufferage to our problem does not yield good results. This is due to the fact that the Sufferage heuristic does not take into account the distribution of computing resources into clusters. We extended the Sufferage heuristic so that sufferage values are computed at the cluster level. For each task, one can compute the MCT of that task on each cluster (over all hosts in the cluster), and the sufferage value can be computed with those cluster-level minima.

Furthermore, the computation of the sufferage values can a little more sophisticated than in the traditional approach. Rather than computing the sufferage value for a task as the difference between the best and second-best achievable completion times, we try to discern groups of clusters that lead to "similar" performance and we define the sufferage value as the average achievable performance difference between the best and second-best cluster groups. We call this heuristic **Extended Sufferage**. We have also tried a new way of computing sufferage (**Sufferage II**) which has proven to be quite efficient in most cases. The basic idea is to introduce an additional level of sufferage computation. Appendix C describes these two heuristics in detail.

Randomization

We also explore techniques that introduce a random factor in some of the heuristics described above. Randomizing is a known technique for scheduling and balancing load in heterogeneous systems [32, 33]. Our experiences proved that introducing some degree of randomness in several of the heuristics for host selection can lead to better scheduling decisions in certain cases. In all the heuristics, it is required to repeatedly chose a task to schedule based on the minimum or maximum of some value (e.g. minimum MCT, maximum MCT, maximum sufferage). It is often the case that multiple task choices achieve or are very close to the required minimum or maximum. The idea is then to pick one of these tasks at random rather than letting this choice be guided by the implementation of the heuristics (e.g. by the way tasks and hosts data structure are stored and ordered, etc.). In order to be fair, we also produced such randomized versions of Min-min, Max-min, and Sufferage.

Summary

To summarize, the heuristics that we consider in our simulations and experiments are:

(H0) Workqueue,

(H1) Min-min,

(H2) Max-min,

(H3) Sufferage,

(H4) Extended sufferage,

(H5) Randomized Min-min,

(H6) Randomized Max-min,

(H7) Randomized sufferage,

(H8) Randomized extended sufferage,

(H9) Sufferage II.

5.6.4 Ordering Input Files

An issue that we have not addressed yet in our description of the scheduling algorithm is the *order* in which input files to a task must be sent. Indeed, since a task needs all its input files present before starting executing, we have complete freedom for choosing the sending order. The strategy we use aims at maximizing file locality. For each file we define its *usage* as the number of tasks that have not started execution and make use of that file. Our scheduler orders the input files transfers by decreasing usage. In other terms, files that have the highest probability of being re-used are sent first. Recall that an input file may not need to be sent at all if its already present on the target remote cluster.

5.6.5 Adaptation to Unexpected Delays

Step (7) of our scheduling algorithm converts the Gantt chart into sequences of sub-tasks for each resource. The implementation of the scheduler is then supposed to go through these sequences in order to launch sub-tasks. According to the task model in Section 5.4 there are some dependencies between sub-tasks, and these dependencies were taken into account when building the Gantt chart, and hence when building the sequences. However, due to unpredicted resource behaviors, it is possible that the sub-task ordering decided when building the Gantt chart precludes the execution of sub-tasks that could be executed immediately. In fact, this should happen only for network links and for output files. Let us clarify this with an example. Say that an output file is scheduled for retrieval on a network link before the transfer of an input file via that link. Now, if the sub-task generating the output file is running late, the network link stays idle when it could send the input file. A simple way of solving this problem would then be to always start the first possible sub-task in a sequence, or at least for file transfers. In our example, it would mean sending the input file before retrieving the output file. However, this modifies the ordering decided by the scheduling heuristics and may have dramatic side effects on the overall run of the application; especially if the task that was blocking the sequence was in fact very close to completion. Our simulation tries both strategies (preserving the order or allowing out-of-order execution) as well as hybrid ones to investigate their impact on the application's makespan.

6 The Parameter Sweep Template

6.1 Target Applications

Our work is motivated by real applications that need to exploit the Grid's computational power to its fullest. We are currently working in collaboration with two different research institutions on two real-world applications. Both these applications fit in our application model and can therefore benefit from our scheduler.

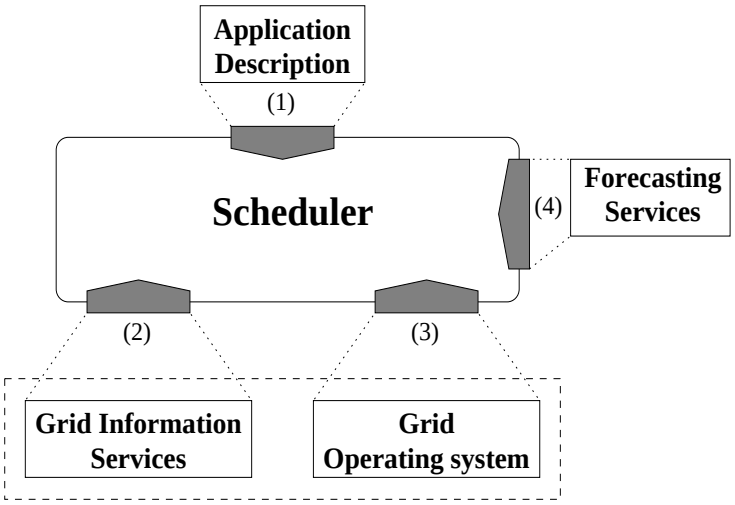


Figure 7: Parameter Sweep Template Implementation

INS2D [35, 34] developed at NASA Ames Research Center is a software application that aims at solving the incompressible Navier-Stokes equations in two-dimensional generalized coordinates for both steady-state and time varying flow. This software can then me used in many fluid mechanics modeling problems (e.g. airfoil modeling at NASA). The code is mostly written in Fortran 77 (with a few C routines) and current INS2D users run on traditional supercomputers (Cray T3E, SGI Origin 2000) or network of workstations. It seems that large-scale runs of INS2D will require the use of many (heterogenous) distributed resources for computation and storage, or in other words, a Grid environment. INS2D does not exhibit any data-sharing patterns among computation task at the moment.

MCell [39, 40], developed at the Salk Institute and the Terry Sejnowski Lab at Cornell University and is a general simulator for cellular microphysiology. MCell uses Monte Carlo diffusion and chemical reaction algorithms in 3D to simulate the complex biochemical interactions of molecules inside and outside of living cells. As a Monte Carlo simulation, MCell must run large numbers of identical, independent simulations for different values of its random number generator seed. Unlike INS2D, MCell tasks usually share large data files and exploiting this pattern is the key to performance in a widely distributed grid.

We expect these two applications to be a good testbed for our first attempt at designing the Parameter Sweep Template. INS2D has a simpler structure than MCell, and this will allow us to see how our scheduling algorithm accommodates different applications. Other applications in other fields of computational science are also being considered. SNP [21], a package for Semi-Non-Parametric time series analysis with many application in economics, will probably be our next target.

6.2 Software Architecture

Figure 7 depicts the general architecture of the first implementation of the PST. The scheduler is the main focus of this work and it is the central piece of our software. It interacts with the outside world via 4 simple interfaces to 4 distinct components. Interface (1) is to the application and its goal is to pass to the scheduler all available information about the application structure (number of tasks, input/output files, user estimates of task execution times, etc.). There are two interfaces to the Grid: interface (2) is to the Grid operating system (for job launching, monitoring, cancellation and data movements) whereas interface (3) is to information services (for Grid topology, number of resources, status of resources,etc.). Finally, interface (4) is to forecasting services that can be used by our scheduler to compute the estimates mentioned in Section 5.2.

This architecture makes it very easy to plug in different components as long as they comply with our interface requirements. In Section 7.1 we explained how we were able to use that architecture for building our simulator by just replacing the actual computational environment by a simulated grid. The following section reviews each of the 4 interfaces and for each describes their current and future implementations.

6.3 Implementation

Interface (1) to the application allows the scheduler to compile all the available information about the user's application. We have defined data structures to describe parameter-sweep applications in a standard way and the interface allows the creation of these data structures. In the current implementation the interface uses a file in a pre-defined format in which the user must enter all application information. As requested by our users, we have also built simple, file-based, application-specific interfaces on top for that general interface (e.g. for MCell). We do not have current plans to build more sophisticated interfaces but, for instance, it would not be difficult to implement graphical Web interfaces.

Interface (2) to the Grid operating system enables our scheduler to launch, monitor and cancel jobs as well as moving data between storage facilities. The interface is defined as a simple API that contains 5 functions. The goal is to implement these functionalities on top of different Grid infrastructure softwares. For early prototyping purposes the first implementation of interface (2) was built directly on top of the socket layer and standard UNIX system calls. We have then developed a version on top of the NetSolve [12] system. Since the interface specification is fairly simple the implementation on top of other Grid software will not be a problem. We are currently designing an implementation on top of the Globus system [19].

Interface (3) to Grid information services enables our scheduler to retrieve information on available resources as well as on past and current measurements on their performance characteristics. Our implementation uses NetSolve's information services directly to gather availability and qualitative information on the resources and the NWS to gather measurements of performance characteristics of hosts and networks. In a scenario where interface (2) is implemented on top of Globus, interface (3) could gather availability and qualitative information from the Metacomputing Directory Service (MDS).

Interface (4) is implemented on top of a forecasting service that is currently available on the Grid: the NWS forecasting modules. Since these modules are implemented as part of NWS, they can easily provide forecasts on NWS measurement data, or be loaded into the scheduler to perform forecast on arbitrary time series. As explained in Section 5.2 we use both possibilities to obtain our running time estimates.

7 Simulation

7.1 Simulator Implementation

We have designed and built a simulator in order to quickly evaluate our different strategies and heuristics. An important feature is that the simulator is completely integrated with the software architecture of the actual template implementation described in Section 6.2. We have seen that the scheduler communicates with other components of the software via simple standard interfaces. One such interface is to the grid system (to launch sub-tasks, check on their completion). Another is to the application (description of task structures, input file sizes). Others are to what we call information services (e.g. NWS) for measurements of available resource performance, or to forecast services (e.g. NWS forecasting modules). One can then write a scheduler and run it in simulation mode by plugging in *simulated* grid and application via the standard interfaces. This is a very convenient design because: (i) once a scheduler has been written and tested in simulation it can be re-used as is for real applications; (ii) different simulations of the grid can be easily implemented as long as they implement the standard interface; (iii) the actual running time of the scheduling algorithm can be realistically evaluated (as opposed to just providing a complexity measure).

The simulator is designed so that arbitrary models can be used for grid information (CPU access delays and loads, network latencies and bandwidths). It is therefore possible to simulate individual parts of the grid information as constant values, random variables sampled from arbitrary distributions, or arbitrary time series (e.g. from NWS measurements on actual systems). Our simulator also makes it possible to simulate scenarios in which hosts or entire clusters are dynamically removed or added to the grid. This allows us to simulate environments where resources are transients.

The description of the simulated application is passed to the scheduler via the standard interface. However, in simulation mode the scheduler receives additional information concerning input and possibly output file sizes (since no actual files exist) as well as information about task-host affinities (see Section 5.2). The following sections describe the current state of the simulator's implementation. Some of the features mentioned in earlier sections are still to be implemented (e.g. computing the deviation from the planned schedule as described in Section 5.6.1). However, the current implementation contains all the elements pertaining to the comparison of the task/host selection heuristics.

7.2 Data-structures Management

The PST software uses four fundamental data-structures (*Disk*, *Host*, *File* and *Work*) as part of its standard interface. These structures are also used by

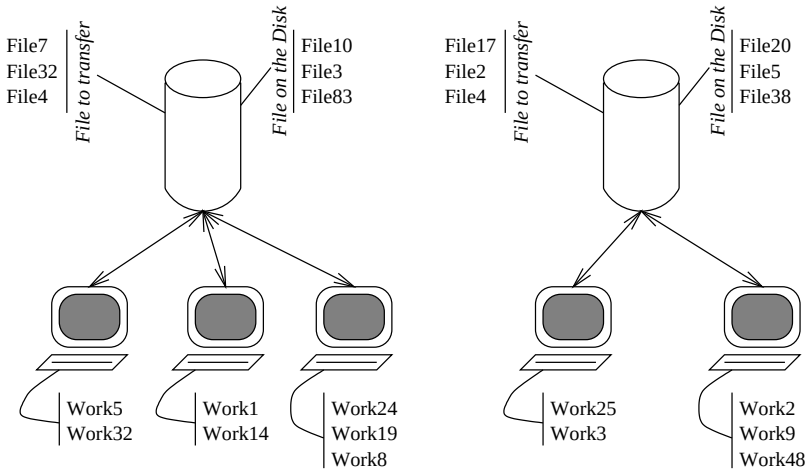


Figure 8: Data-structures

the simulator (see Figure 8). They are used to keep track of the state of the application and of the grid. Note that the origin is connected to a cluster with a single network link and that there is a single storage device per cluster. In the current implementation of our simulator, we can therefore use a single data structure, *Disk*, to describe both the storage device and the network link. In what follows, we will use "*Disk*" to denote either the cluster, the network link or the storage device. Sections 7.2.1 and 7.2.2 describe how one can define virtual grids and applications for the purpose of simulation. In this section we are only concerned with the four fundamental data structures. A lot of information is stored in these structures (e.g. URLs, IP addresses, etc.). Basically, a *Disk* is described by a name, a list of *Files* physically present on the *Disk* and a list of *Hosts* that can access files on the *Disk*. Each *Disk* maintains a FIFO of *Files* to be transfered. These FIFOs are manipulated by the scheduler. A *Host* maintains a list of *Disk* it can access (in our representation, we need each *Host* to access only one *Disk* to ensure a real clusterized architecture). As for a *Disk*, a *Host* maintains a FIFO of *Works* to be executed on that *Host*. A *Work* represents an application task and consists of a list of input *Files*, a list of output *Files* and additional information that is irrelevant for the simulation, such as location of the executables for each type of architecture for instance. *Files* are described by a name, a size and a list of *Work* that use or produce them. All four structures possess meta-data describing their behaviors (CPU-load for *Hosts*, latency and bandwidth for *Disks*), their running time estimates on different architectures (for *Works*) or estimated sizes (for output files).

7.2.1 Grid description files

A Grid description file describes events that are going to happen during a run of the application, such as host failures or changes in network behavior. This file is parsed in order to create a list of events that is used during the simulation. To each resource (network link or host) in the simulated grid is associated a *symbolic name* for identification purposes. Furthermore, the behavior of each

resource (in terms of deliverable performance) is described in a separate file, a *behavior file*. A behavior file contains a circular time series of values (possibly tuples) that represent the evolution of the resource's performance throughout time. In our current implementation and experiments we have chosen a 5 second interval in between values (but this is easy to customize).

Link Behavior A file describing the behavior of a Link starts with a header that specifies the number of latency/bandwidth tuples present in this file. An infinite loop through these values simulates the behavior of the link. The latency is given in milliseconds and the bandwidth in byte per second.

Host Behavior A file describing the behavior of a *Host* starts with a header that specifies the number of values for CPU availability present in this file. An infinite loop through these data simulates the behavior of the *Host*. The values are in percentage.

Using such a structure for behavior files allows the simulator to be very flexible. Indeed, simulating stable resources can be done by creating a file with one single value (since the time-series are viewed as circular). One can also simulate resources whose performance values are sampled from some given distribution by putting such samples directly in a behavior file. Lastly, one can create a behavior file with historical performance data measured on real resources. The NWS can provide such data. Let us now describe the syntax for each event in a simulated grid:

Events Files describing the grid start with a header that specifies a software version number. There are six possible events with the following syntax:

- <*time*>:`ADD_CLUSTER` <*cluster_name*> <*behavior_file_name*> <*offset*>: At time *time*, a cluster with symbolic name *cluster_name* is added to the grid. The bandwidth and latency for this cluster are simulated according to the time-series in file *behavior_file_name*. The *offset* is the index of the starting point in the time-series. This allows to use the same behavior file for different resources while introducing some de-synchronization among these resources.
- <*time*>:`ADD_HOST` <*cluster_name*> <*host_name*> <*behavior_file_name*> <*offset*> <*ARCH_TYPE*>: At time *time*, a *Host* with the symbolic name *host_name* is added to the cluster named *cluster_name*. The CPU availability for this *Host* is taken from file *behavior_file_name* and one can also specify an offset. *ARCH_TYPE* is the architecture type for that *Host*. It is thus possible to simulate task/architecture affinities by introducing hosts with different architecture types in the grid.
- <*time*>:`REMOVE_CLUSTER`<*cluster_name*>: At time *time*, the cluster named *cluster_name* is removed from the grid and any *Host* belonging to this cluster shutdowns. Tasks running on these *Hosts* fail and files in that cluster's storage are unavailable until the cluster comes back up.

- <*time*>:`REMOVE_HOST` <*host_name*>: At time *time*, the *Host* named *host_name* is removed from the grid and the tasks running on that *Host* are considered to have failed.
- <*time*>:`CHANGE_HOST_BEHAVIOR` <*host_name*> <*behavior_file_name*> <*offset*>: At time *time*, the behavior of *Host host_name* changes to one that is described by file *behavior_file_name*. As usual, one can also specify an offset.
- <*time*>:`CHANGE_CLUSTER_BEHAVIOR` <*cluster_name*> <*behavior_file_name*> <*offset*>: At time *time*, the behavior of the network link to cluster *cluster_name* changes according to file *host_file_name*. One can specify an offset.

Events are sorted by *time* and blank lines or lines beginning with '`#`' in the grid description file are ignored so that comments can be inserted. Being able to change the behavior of a *Host* or of a network link makes it possible to simulate sudden network contention or CPU load in order to test the scheduling algorithm's robustness to sudden changes. An example of grid description file is given in appendix A.

7.2.2 Application description files

A file describing an application starts with a header that specifies a software version number, a number of *Files* (*file_number*) and a number of tasks (*task_number*). That header is followed by *file_number* file description lines and *task_number* task description lines. A file description lines has the following syntax: `File:` <*File_id*> <*File_name*> <*real_size*> <*estimated_size*>. *File_id* is a unique number comprised between 0 and *file_number* -1; *File_name* is a symbolic name for the file (not used by the simulator but convenient for the user); *real_size* is the size of the file (used in the simulator). But as exact file sizes might not necessarily be known (for instance for output files), it is possible to give an estimated size that will be used in the scheduler. This makes it possible to study the effect of inacurrate or incomplete information on the schedule. A task description line follows the following syntax: `Work:` <*Work_id*> <*input* : $File_id_1$ ... $File_id_{input}$> <*output* : $File_id_1$ ... $File_id_{output}$> <*ARCH* : run_time_1 ... run_time_{ARCH}> <*ARCH* : $est_run_time_1$ ... $est_run_time_{ARCH}$>. *Work_id* is a unique number comprised between 0 and *task_number* -1. Other parameters are respectively the input files, the output files, the running time on each different architecture (used in the simulator) and the user's estimated running time on each different architecture (used in the scheduler). The only line ordering requirement is that a file must be declared before it is used by a task. As before, comments might be added thanks to a '`#`' at the beginning of a line and blank lines are ignored. An example of an application description file is provided in Appendix B.

7.3 Event-driven scheduler

The interaction between the scheduler and the Grid software is in some sense "bi-directional". On the one hand, the scheduler can initiate computations or file transfers within the Grid by calling a set of functions (the `do_*` functions that are the object of Section 7.3.4). On the other hand, the scheduler can be

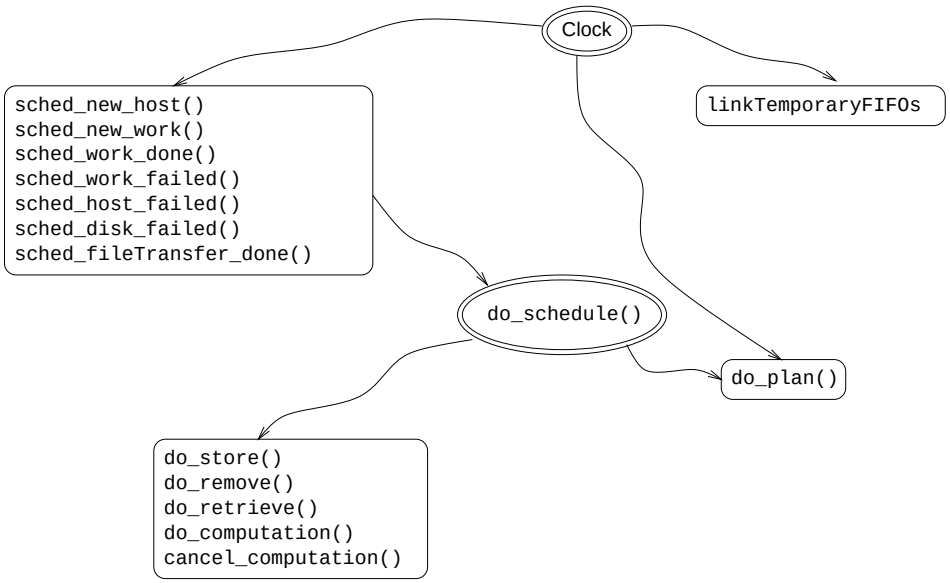


Figure 9: Scheduler structure

notified of Grid events by calls to another set of functions (the `sched_*` functions, Section 7.3.2). It is the responsibility of the PST software to interface to the Grid operating system to capture those events and call the appropriate functions (in simulation mode or not). Our scheduler is an event-driven scheduler: the scheduler is called only when some particular events occurs (an host failure, the end of a computation or of a file transfer, ...). The core of the scheduler consists of two functions, `do_plan()` and `do_schedule()`. `do_plan()` has already been introduced and is in charge of making long term planning for the schedule of the application. Each time that function is called it takes into account changes in the grid and application states in order to refine the planned schedule. The implementation of `do_plan()` is described in detail in Section 7.4. `do_schedule()` reacts to events and takes into account the long-term planning generated by `do_plan()` to assign tasks to resources. However, since `do_plan()` computes its long-term schedule with possibly inaccurate predictions on resource performance, it is not practical to try to follow that exact schedule. For instance, it may be that a resource becomes unexpectedly unloaded and is able to get ahead of the schedule. `do_schedule()` is in charge of coping with such behavior and the hope is that `do_plan()` will be called again in order to adapt to changing conditions. `do_plan()` fills in the *Host*'s and *Disk*'s FIFOs we mentioned in Section 7.2 and `do_schedule()` pulls out tasks from these FIFOS for execution. The structure of the scheduler is depicted in Figure 9 and a detailed description is given in the following sections.

7.3.1 Structure

Our simulator makes progress according to a global "clock". Each event in the grid description file is supposed to happen at a given time (see Section 7.2.1). When the global clock reaches that time, a call to an appropriate `sched_*`

function is generated. Some of these functions call `do_schedule()` which may assign file transfers to free links or assign computations by calling one of the `do_*` functions. All these functions are described in following sections.

There are two ways of calling `do_plan()`. As seen in Figure 9 it can be called at given times (see Section 5.3). It is also possible for `do_schedule()` to call `do_plan()` directly if it deems it necessary. For instance, it may be that a a new cluster becomes available in the grid or that a host becomes idle unexpectedly and that its computational power stays untapped until `do_plan()` is called. Deciding when `do_schedule()` should call `do_plan()` is not entirely straightforward. For instance, some scheduling heuristics may decide to leave some hosts unused on purpose. The hope is that if the $\{t_i\}$ series is well chosen there should be no need for `do_schedule()` to call `do_plan()` unless new hosts become available.

Finally, we mentioned earlier that our simulator takes into account the execution time of `do_plan()`. Indeed, since the heuristics implemented in `do_plan()` may be computationally expensive, it seems more realistic to let the application make progress while the schedule is being computed. In that view, an additional parameter to the simulator is the average simulated execution time of `do_plan()`. This is implemented in the following way. Just before a call to `do_plan()`, each *Host* and *Disk* is assigned a temporary FIFO that is filled with a few *Works* or *Files* that are to be performed while the schedule is being computed. These FIFOs are merged with the real ones (the ones used by `do_schedule()`) when the clock reaches `do_plan()`'s simulated completion time.

7.3.2 `sched_*` functions

There are seven `sched_*` functions that are part of the scheduler and must be called by the PST software when particular events occur.

- `int sched_new_host(HostList host)`: This function makes the scheduler aware of a new host(s) in the system that is available for computation. New *Hosts* are moved to a global list of available *Hosts*. This function calls `do_schedule()` in order to assign work to a *Host* as soon as it is ready.

- `int sched_new_work(WorkList work)`: This function gives more work units to the scheduler. New *Hosts* are moved to a global list of *Works* to be done. This function does not need to call `do_schedule()` since adding new wotk to the application should have little impact on the tasks currently running. For our current target applications (see Section 6.1) all tasks are known a-priori and this function is only called once.

- `int sched_work_done(WorkList work)`: When the transport mechanism determines that some work unit(s) have successfully completed their computation, it must call this function to notify the scheduler. *Works* given in parameter are moved from a global list of running *Works* to a global list of completed *Works*. `do_schedule()` is called by this function since some *Hosts* are now available.

- `int sched_host_failed(HostList host)`: When the transport mechanism determines that some hosts in the system have shutdown or have

become unreachable, it must call this function to notify the scheduler. *Hosts* are moved to a global list of failed *Hosts*. If a *Work* was running on this *Host*, `sched_work_failed()` is called. Each *Host*'s FIFO is emptied and *Works* are moved to a list of *Works* to do. This function doesn't need to call `do_schedule()` (since tasks are independent).

- `int sched_work_failed(WorkList work)`: When the transport mechanism determines that some work unit(s) has failed, it must call this function to notify the scheduler. As all tasks are independent, the *Work* is just postponed until a call to `do_plan()`. In our case only `sched_host_failed()` calls `sched_work_failed()`. There is no need to call `do_schedule()` since in our current model the failure of a *Work* might only be due to the failure of an *Host*.

- `int sched_disk_failed(DiskList links)`: When the transport mechanism determines that some *Disk* (i.e. cluster) in the system has shutdown, it must call this function to notify the scheduler. *Disks* are moved to a global list of failed *Disks*. Each *Host* connected to that *Disk* (i.e. in that cluster) is considered to have failed and `sched_host_failed()` is called for each host. The FIFO of *Files* to transfer to that *Disk* is emptied. This function doesn't need to call `do_schedule()` (once again, it is due to the independence of the tasks).

- `int sched_fileTransfer_done(File *file, Disk *link);` When the transport mechanism determines that a file transfer *Disk(s)* has completed successfully, it must call this function to notify the scheduler. *Disks* are moved from a global list of busy *Disks* to a global list of available *Disk*. This function calls `do_schedule()` in order to assign file transfers to a *Disk* as soon as it is available.

7.3.3 `do_schedule()`

This function follows the algorithm described in Figure 10. As an optional feature, it is possible to allow `do_schedule()` to perform Work Stealing within a Cluster (line 7-8 of Figure 10's algorithm). Assignments of file transfers and computations are done by calling one of the `do_*()` functions. In Section 5.6.5 we described the fact that there are different ways of handling scheduled file transfers over a network link. The algorithm as it is written in Figure 10 uses a greedy strategy (this is done in steps (12-13)). We have not yet experimented with other strategies.

7.3.4 `do_*` functions

These functions are part of the PST software since they enable the scheduler to interact with the grid:

- `char *do_store(Disk *disk, File *file)`: This function initiates the transfer of a *File* to a *Disk* and returns a capability.

- `int do_remove(char *capability)`: This function removes a *File* from a *Disk* given a capability.

```
    do_schedule(){
(1)     foreach currently available Host
(2)         if there is a Work in the head of its FIFO
(3)             if this Work is ready
(4)                 remove the Work from the head of the FIFO
(5)                 assign the computation of this Work to this Host
(6)         else
(7)             if another Host on the same cluster has a ready Work to do
(8)                 assign it to the available Host
(9)             calldoplan = 1
(10)    foreach currently available Disk
(11)        if there is a File to transfer in the head of its FIFO
(12)            while this File is a non ready outputFile
(13)                try the next File in the FIFO
(14)            if  this File is a ready File
(15)                remove the File from the FIFO
(16)                assign the transfer of this File to this Disk
(17)    if  (calldoplan == 1) call do_plan()
(18)    }
```

Figure 10: Scheduling algorithm

- `int do_retrieve(Work *work, Host *host)`: This function initiates the transfer of the output *Files* to a *Work* that was scheduled on a given *Host*.
- `int do_computation(Work *work, Host *host)`: This function initiates a *Work*'s computation on a *Host*.
- `int cancel_computation(Work *work, Host *host)`: This function cancels a *Work*'s computation. This will be useful in case we allow task duplication.

7.4 Planning

This section aims at presenting the different steps in our implementation of the `do_plan()`function and the different aspects of this implementation that might need tuning for future developments. `do_plan()`is implemented as a sequence of the 5 following steps:

- Truncating the FIFOs: When `do_plan()`is called, it is generally the case that *Hosts* and *Disks* still have a fair amount of sub-tasks that had been scheduled but have not started execution. This is due to a conservative choice for the value of Δ (see Section 5.5). These remaining sub-tasks should be removed from the *Hosts*' and *Disks*' FIFOs so that they can be reconsidered for scheduling. However, since we simulate the fact that `do_plan()`does not execute instantaneously (Section 3), we must leave a few sub-tasks in those FIFOs. This is a way to ensure that `do_plan()`'s execution can be overlapped with actual compuattions and file transfers. The

current implementation truncates the *Hosts*' FIFOs to a a fixed number of *Work* and leaves in *Disk*' FIFOs only *Files* that are used or generated by those *Works*.

- Task set selection: We explained in Section 5.6.2 that it is often desirable to select a subset of the tasks that have not started execution for scheduling. Among those tasks, our scheduling algorithm will only schedule enough to "fill" the Gannt chart appropriately. This selection might be done using affinity between different tasks in terms of file sharing for instance. In the current implementation, we do not select any subset but instead use the entire set of tasks that remain to be executed. We schedule a number of task proportional to the number of available *Hosts* in an attempt at filling the Gantt chart evenly.

- Gantt Chart creation: The core of our scheduler uses a structure named `linkPlan`. It consists of a table (*Hosts* $\times$ *Works*) and of FIFOs (one for each resource, i.e. *Disk* or *Host*) in which time-stamped sub-tasks can be inserted. The table is used for keeping track of completion times and the FIFOs form a GanttChart. The *Works* and *Files* that have been left in the *Host*' and *Disks*' FIFOs are inserted in the Gannt chart (i.e. the FIFOs of the `linkPlan` structure).

- Heuristic: A typical heuristic takes in input a `linkPlan` structure, the number of tasks to schedule and returns a filled-in Gantt chart.

- Updating FIFOs: The content of the Gantt chart is used to add *Works* and *Files* to the *Hosts*' and *Disks*'s FIFOS.

8 Some Early Experimental Results

The simulator generates a consequent amount of results. Indeed, simulating 10 task/host selection heuristics on 4 different grids for 4 different applications leads to 160 makespan measurements. Since we aim at simulating and analyzing many more configurations it became quickly necessary to design some framework for obtaining, gathering, exploring, and plotting simulation results. In that view, we wrote a series of Perl scripts (for a total of 1,500 lines). One script is in charge of assisting the user for generating application description files. For instance, the script makes it easy to create regular file access patterns (e.g. subsets of tasks sharing a same input file), to generate non-random or random (with precise probability distributions) file sizes and task execution time, to perform random perturbations in the file access patterns, or to modify the computation/communication ration (CCR) of the application. Another script is in charge of running the simulator and storing the results inside a structured database. This is fairly convenient as the simulator's user needs not be concerned with avoiding running the same simulation multiple times, keeping track of the results, etc. Lastly, a set of scripts can be used to easily extract and sort database entries and obtain corresponding performance graphs. One such graph is shown in Figure 11.

The grid environment simulated for obtaining the results in Figure 11 constists of three clusters. All hosts in the system are identical but with different load conditions. The clusters respectively contain four 100% available hosts, six

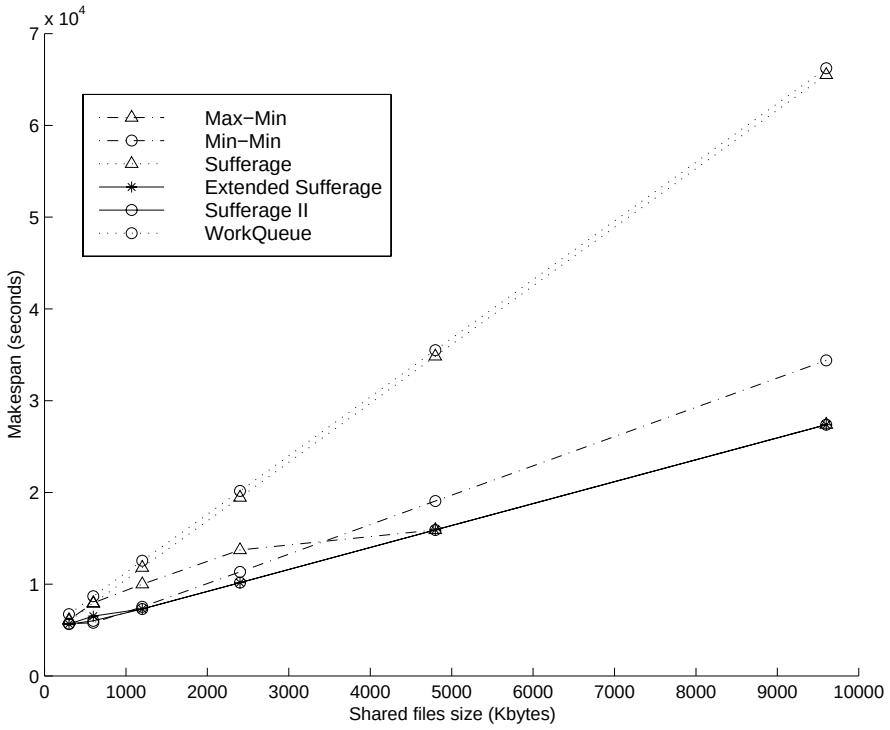


Figure 11: Effect of shared file size on different scheduling strategies

70% available hosts, and twelve 30% available hosts. All network links from the origin to the cluster are identical and deliver a constant 10 Kbyte/s bandwidth. There are not host failures during the run of the application. This sytem is not very realistic since there is no CPU or network load fluctuations. However, we believe that such systems provide adequate initial guidance for the understanding of scheduling algorithms' behaviors. Future work will include experiments with more realistic grid environments. The application we simulated consists of 400 tasks subdivided into 8 *chunks* of 50 tasks. Tasks within the same chunk all share one large input file and all tasks of the application take two non-shared input files. Each tasks produces one output file. Tasks' computation times in seconds are uniformely distributed on the interval [90,110]. The sizes of the un-shared input file and output files in Kbytes are uniformly distributed on the interval [8,12]. This simulation assumes perfectly accurate running time estimates for all sub-tasks and files are transfered on network links on a first available basis (see Section 5.6.5). The graph is obtained by simulating six heuristics for increasing sizes of the shared input files.

The first observation is that the workqueue (H0) leads to very bad performance because it cannot take advantage of the input file sharing within chunks of tasks in the application. The same is true of the basic version of Sufferage (H3) since it computes sufferage values at the host level and not at the cluster level (see Section 5.6.3). All curves are pseudo-linear but for the Max-min heuristic (H2). The heuristics leading to best performance for all file sizes are Extended Sufferage (H4) and Sufferage II (H9). The Min-min heuristic (H1)

fails to capture all file access patterns and hence leads to worse performance than sufferage-based ones.

Let us note to interesting facts about that example. First, randomized heuristics (H5), (H6), (H7), (H8) led to identical performance as their non-randomized counterparts. Proving or disproving the usefulness of randomzation in our framework will be the object of future work. Second, Max-min (H2) exhibits a peculiar behavior. For shared file sizes 4800 Kb and 9600 Kb it yields the same performance as the sufferage-base heuristics. The aforementioned Perl scripts allow in-depth exploiration of the database and in particular a complete description of each application's run. Examining such data for the last two runs of Max-min reveals that the good performance of its schedule is due to a special idiosyncrasis of the application's structure vs. the grid's topology. In other words, that behavior cannot be observed in the general case. Such a phenomenon shows that care must be taken when interpreting simulation results and that only a wide variety of results will prove meaningful when comparing heuristics. In this example, changing a few parameters in the grid description file would cause Max-min's performance cureve to be linear, somewhere in between Min-min's and Sufferage's curve and this seems to be the general case.

Even though the purpose of this document is not to present new results concerning scheduling heuristics but rather to describe our initial models and the framework that will enable us to push research further, these first results are encouraging and indicate that simulation can indeed provide insight when developing a scheduling algorithm for the Grid.

9 Conclusion

The purpose of our research is to evaluate and compare different scheduling strategies for a given class of applications, i.e. parameter-sweeps, in grid environments. Simulation seemed the appropriate way to tackle this problem as (i) running real applications on real environments is extremely time consuming and wastes computational and network resources; (ii) real environment make it very difficult to obtain reproducible and meaningful comparison results; (iii) real environments are not as parameterizable as simulated ones. Our first implementation of a simulator was detailed in Section 7 and has been used to obtain initial results. Some of these results were presented in Section 8 and they provide elements for comparison of different task/host selection heuristics. The two basic lessons learned there were that (i) workqueue strategies are not well adapted to our applications when there are file-sharing patterns; (ii) sufferage-based heuristics are more adapted to the exploitation such file-sharing patterns. Those are only our very first experiments and there are countless possibilities, in terms of grid environments and application structures, for further simulations. Obtaining new results and providing meaningful interpretations are going to be one of the focuses of our upcoming research.

There are many ways in which the simulator itself can be extended. We already mentioned that some features described in Section 5 were not implemented in the simulator as of now. Implementing those features will make it possible to study the impact of various factors on the task/host selection scheduling heuristics. These factors include but are not limited to quality of information (see Section 5.2.2), schemes for choosing intervals between scheduling events (see

Section 5.6.1), short-term ordering of output file transfers on network links (see Section 5.6.5).

Once we have accumulated and anlysed enough experimental results to draw general conclusions and rules concerning the use of different heuristics in different environments, we will be able to perform experiments in real-world grids with real applications (such as the ones highlighted in Section 6.1). We have also started giving some thought to new task/host scheduling heuristics that strive to "understand" the structure of the application a little more in depth. The challenge is to manage a good trade-off between computational complexity, effectiveness, and sensitivity to bad quality of information. These investigations will be the object of a future document.

References

[1] `http://www.hsdal.ufl.edu/Projects/Osculant/`.

[2] `http://www.activetools.com`.

[3] D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, and R. Sutherst. The ,Nimrod Computational Workbench: A Case Study in Desktop Metacomputing. In *Proceedings of the 20th Australasian Computer Science Conference*, Feb. 1997.

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

[5] P. Arbenz, W. Gander, and M. Oettli. The Remote Computational System. *Parallel Computing*, 23(10):1421–1428, 1997.

[6] M. Beck, J. Plank, T. Moore, and W. Elwasif. Why IBP Now. *The International Journal of Supercomputer Applications and High Performance Computing*, to appear.

[7] F. Berman. *The Grid, Blueprint for a New computing Infrastructure*, chapter 12. Morgan Kaufmann Publishers, Inc., 1998.

[8] F. Berman and R. Wolski. The AppLeS Project: A Status Report. In *Proc. of the 8th NEC Research Symposium, Berlin, Germany*, May 1997.

[9] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Proc. of Supercomputing'96, Pittsburgh*, 1996.

[10] F. Berman, R. Wolski, and G. Shao. Performance Effects of Scheduling Strategies for Master/Slave Distributed Applications. Technical Report TR-CS98-598, U. C., San Diego, 1998.

[11] R. D. Blumofe and C. E. Leiserson. Multithreaded Computations by Work Stealing. In *Proc. of Ann. Symp. on Founcdations of Computer Science*, pages 356–368, Nov. 1994.

[12] 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.

[13] H. Casanova, M. Kim, J. S. Plank, and J. Dongarra. Adaptive Scheduling for Task Farming with Grid Middleware. *The International Journal of Supercomputer Applications and High Performance Computing*, 1999. to appear.

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

[15] G. Djordjevič and M. Tosič. A heuristic for scheduling task graphs with communication delays onto multiprocessors. *Parallel Computing*, 22:1197–1214, 1996.

[16] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. In *Proceedings of SuperComputing'99*, 1999. accepted for publication.

[17] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings or the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, 1997.

[18] 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.

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

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

[21] A. Ronald Gallant and G. Tauchen. SNP: A Program for Nonparametric Time Series Analysis. Duke economics Working Paper #95-26, Duke University, 1997. v8.6 (revised 1997).

[22] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Jr. Reynolds. A Synopsis of the Legion Project. Technical Report CS-94-20, Department of Computer Science, University of Virginia, 1994.

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

[24] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel 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.

[25] 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.

[26] M. A. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple DAGS in a distributed heterogeneous environment. In *7th IEEE Heterogeneous computing Workshop (HCW'98)*, pages 70–78, Mar. 1998.

[27] C. Leangsuksun, J. Potter, and S. Scott. Dynamic task mapping algorithms for a distributed heterogeneous computing environment. In *4th IEEE Heterogeneous computing Workshop (HCW'95)*, pages 30–34, Apr. 1995.

[28] M. Litzkow and M. Livny. Experience with the Condor Distributed Batch System. In *Proc. of IEEE Workshop on Experimental Distributed Systems*, pages 97–101. Department of Computer Science, University of Winsconsin, Madison, 1990.

[29] M. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th International Conference of Distributed Computing Systems*, pages 104–111. Department of Computer Science, University of Winsconsin, Madison, June 1988.

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

[31] 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. to appear.

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

[33] M. Mitzenmacher. On the Analysis of Randomized Load Balancing Schemes. *Theory of Computer Systems*, 32:361–386, 1999.

[34] S. Rogers. A Comparison of Implicit Schemes for the Incompressible Navier-Stokes Equations with Artificial Compressibility. *AIAA Journal*, 33(10), Oct. 1995.

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

[36] H.G. Rotithor. Taxonomy of synamic task scheduling schemes in distributed computing systems. *Proceedings on Computer and Digital Techniques*, 141(1):1–10, Jan. 1994.

[37] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proc. of Sym. on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[38] 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.

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

[40] J.R. Stiles, D. Van Helden, T.M. Bartol, E.E. Salpeter, , and M.M. Salpeter. Miniature end-plate current rise times <100 microseconds from improved dual recordings can be modeled with passive acetylcholine diffusion form a synaptic vesicle. In *Proc. Natl. Acad. Sci. U.S.A.*, volume 93, pages 5745–5752, 1996.

[41] 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)*, Aug 1999. to appear.

[42] H. Topcuoglu, S. Hariri, and M. Wu. Task Scheduling Algorithms for Heterogeneous Processors. In *Heterogeneous computing Workshop (HCW'99)*, pages 3–14, Apr. 1999.

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

A An Example of Grid Description File

The following file describes a grid that contains 6 clusters with transient hosts.

```
GridDescription 1
0:ADD_CLUSTER <cluster3> <stable_100Kb> <5
0:ADD_HOST <cluster3> <host3.1> <stable_50> <1> <0>
0:ADD_HOST <cluster3> <host3.2> <stable_50> <0> <1>
0:ADD_CLUSTER <cluster4> <stable_100Kb> <12>
0:ADD_HOST <cluster4> <host4.1> <stable_70> <2> <0>
0:ADD_HOST <cluster4> <host4.2> <stable_50> <0> <1>
0:ADD_CLUSTER <cluster5> <stable_100Kb> <13>
0:ADD_HOST <cluster5> <host5.1> <stable_90> <4> <0>
0:ADD_HOST <cluster5> <host5.2> <stable_100> <2> <1>
0:ADD_CLUSTER <cluster6> <stable_100Kb> <12>
0:ADD_HOST <cluster6> <host6.1> <stable_100> <0> <0>
0:ADD_HOST <cluster6> <host6.2> <stable_100> <0> <1>
200:REMOVE_CLUSTER <cluster3>
553:CHANGE_HOST_BEHAVIOR <host4.2> <loaded_2> <4>
1000:ADD_CLUSTER <cluster1> <stable_10Kb> <2>
1000:ADD_HOST <cluster1> <host1.1> <stable_70> <4> <0>
1000:ADD_HOST <cluster1> <host1.2> <stable_70> <3> <1>
1000:ADD_CLUSTER <cluster2> <stable_1Mb> <0>
1000:ADD_HOST <cluster2> <host2.1> <stable_50> <4> <0>
1000:ADD_HOST <cluster2> <host2.2> <stable_90> <2> <1>
1500:REMOVE_CLUSTER <cluster4>
1550:CHANGE_CLUSTER_BEHAVIOR <cluster3> <stable_100Kb> <2>
```

B An Example of Application Description File

The following file describes and application that consists of 12 tasks.

```
WorkDescription 1 22 12
# Input files
File: <0> <file_0>    <10456>   <10456>
File: <1> <file_1>    <22356>   <22356>
File: <2> <file_2>    <1230456> <1230456>
File: <3> <file_3>    <234559>  <234559>
File: <4> <file_4>    <1130456> <1130456>
File: <5> <file_5>    <1230456> <1230456>
File: <6> <file_6>    <230456>  <230456>
File: <7> <file_7>    <1334559> <1334559>
```

```
File: <8> <file_8>    <120456>  <120456>
File: <9> <file_9>    <1223456> <1223456>
# Output files
File: <10> <file_10>  <230456>  <230456>
File: <11> <file_11>  <323459>  <323459>
File: <12> <file_12>  <230456>  <230456>
File: <13> <file_13>  <334559>  <334559>
File: <14> <file_14>  <230456>  <230456>
File: <15> <file_15>  <234559>  <234559>
File: <16> <file_16>  <230456>  <230456>
File: <17> <file_17>  <323459>  <323459>
File: <18> <file_18>  <230456>  <230456>
File: <19> <file_19>  <334559>  <334559>
File: <20> <file_20>  <230456>  <230456>
File: <21> <file_21>  <234559>  <234559>
# Tasks
Work: <0> < 3 : 0 1 2> < 1 : 10 > < 3 : 600 500 600 > < 3 : 610 503 602 >
Work: <1> < 3 : 1 2 3> < 1 : 11 > < 3 : 600 500 600 > < 3 : 590 502 604 >
Work: <2> < 3 : 3 4 5> < 1 : 12 > < 3 : 700 600 600 > < 3 : 703 604 623 >
Work: <3> < 3 : 2 6 8> < 1 : 13 > < 3 : 300 200 600 > < 3 : 304 206 601 >
Work: <4> < 3 : 5 8 9> < 1 : 14 > < 3 : 500 300 600 > < 3 : 502 308 578 >
Work: <5> < 3 : 7 2 3> < 1 : 15 > < 3 : 300 200 600 > < 3 : 303 210 600 >
Work: <6> < 3 : 4 1 2> < 1 : 16 > < 3 : 600 500 600 > < 3 : 610 503 592 >
Work: <7> < 3 : 9 2 6> < 1 : 17 > < 3 : 600 500 600 > < 3 : 612 512 600 >
Work: <8> < 3 : 2 4 9> < 1 : 18 > < 3 : 700 600 600 > < 3 : 689 603 620 >
Work: <9> < 3 : 2 6 8> < 1 : 19 > < 3 : 300 200 600 > < 3 : 256 202 577 >
Work: <10> < 3 : 3 8 5> < 1 : 20 > < 3 : 500 300 600 > < 3 : 512 301 609 >
Work: <11> < 3 : 6 2 3> < 1 : 21 > < 3 : 300 200 600 > < 3 : 300 200 600 >
```

C Task/Host Selection Heuristics

Let us define some terminology that will be used to describe the different heuristics. We denote by $\mathrm{Compl}(T_i, H_{j,k})$ the estimated completion time of task T_i on host $H_{j,k}$t. We also introduce the argmin operator defined as follows:

Definition C.1 *Given a function f from $\mathbb{R}^n$ into $\mathbb{R}$, we define* $\mathrm{argmin}_{x\in\mathbb{R}^n} f(x)$ *such that:*

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

Note that the operator only denotes one of the possible vectors that achieves the minimum of the function f. The choice of that vector is then left to the implementation. Randomizing that choice leads to the randomized versions of the heuristics as discussed in 5.6.3. For each heuristic, we describe how a task/host assignment is chosen among all possible assignments.

Min-min : For each task T_i, the host that achieves the MCT is $H_{c_i^{(1)},h_i^{(1)}}$, with:

$$(c_i^{(1)}, h_i^{(1)}) = \mathrm{argmin}_{j,k}(\mathrm{Compl}(T_i, H_{j,k})),$$

then, one determines the task that has the lowest MCT, say T_s as:

$$s = \text{argmin}_i(\text{Compl}(T_i, H_{c_i^{(1)},h_i^{(1)}})).$$

Task T_s is then assigned to host $H_{c_s^{(1)},h_s^{(1)}}$.

Max-min : Similarly to Min-min, on can write:

$$(c_i^{(1)}, h_i^{(1)}) = \text{argmin}_{j,k}(\text{Compl}(T_i, H_{j,k})),$$

and

$$s = \text{argmax}_i(\text{Compl}(T_i, H_{c_i^{(1)},h_i^{(1)}})).$$

Task T_s is then assigned to host $H_{c_s^{(1)},h_s^{(1)}}$.

Sufferage : For each task T_i, the hosts that achieve the best and second-best completion time are $H_{c_i^{(1)},h_i^{(1)}}$ and $H_{c_i^{(2)},h_i^{(2)}}$, with:

$$(c_i^{(1)}, h_i^{(1)}) = \text{argmin}_{j,k}(\text{Compl}(T_i, H_{j,k})),$$
$$(c_i^{(2)}, h_i^{(2)}) = \text{argmin}_{j \neq c_i^{(1)}, k \neq h_i^{(1)}}(\text{Compl}(T_i, H_{j,k})).$$

The sufferage value of task T_i is then defined as:

$$\text{Suff}_i = \text{Compl}(T_i, H_{c_i^{(2)},h_i^{(2)}}) - \text{Compl}(T_i, H_{c_i^{(1)},h_i^{(1)}}).$$

Note that the sufferage value can be equal to 0 if the MCT can be achieved on more than one host. The task T_s with the maximum sufferage value is such that:

$$s = \text{argmax}_i(\text{Suff}_i).$$

Task T_s is then assigned to host $H_{c_s^{(1)},h_s^{(1)}}$.

Extended Sufferage : For each task T_i one can compute its estimated MCT on cluster C_j, say m_j^i, as:

$$m_j^i = \min_k \text{Compl}(T_i, H_{j,k}).$$

A natural way of computing a sufferage value would be to take the difference between the best and second-best m_j^i. However, we propose here an extended way for computing the sufferage.

Consider figure 12 where we plot the **ordered** cluster-level MCTs for two tasks, (a) and (b). Instead of just limiting our analysis to the first two MCTs we take a broader look at all the MCTs for all clusters. Intuitively we define the sufferage value as the height of the first "jump" in the cluster-level MCT curve. In the example of Figure 12, priority would be given to task (b). In practice, we detect the jump by computing the approximate derivative of the MCT curve, computing its mean and standard deviation. We define the first jump as the first point where the value of the derivative is larger than its mean

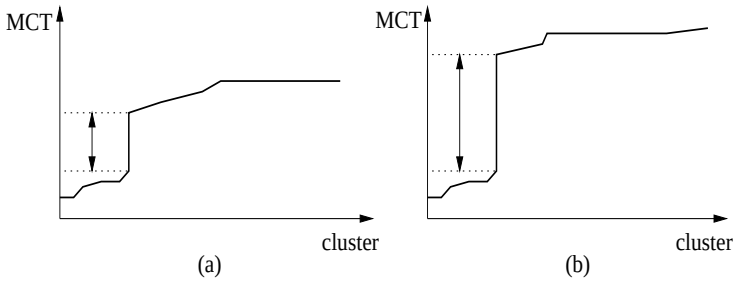


Figure 12: Extendend Sufferage

plus one standard deviation. We denote by Suff_i the height of the first jump. This way of extending the notion of sufferage is intended to capture more of the file access patterns in our application and its effectiveness is demonstrated in simulation and experiments. As for the regular sufferage, the task T_s with

$$s = \text{argmax}_i(\text{Suff}_i).$$

is assigned to the host that achieves the MCT.

Sufferage II : Consider figure 13 where we plot the **ordered** cluster-level MCTs for two tasks, (a) and (b).

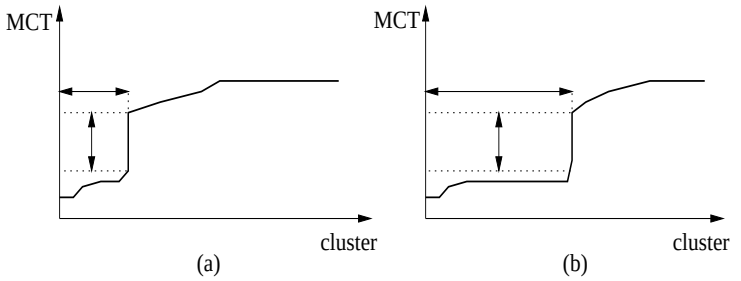


Figure 13: Sufferage II

The idea here is to give priorities to tasks whose first jump is the closest to the origin. In other terms, to tasks with the smallest number of clusters that lead to "good" performance. For the two tasks shown on Figure 13, task (a) should be given priority since it has the least leeway in terms of choosing a "good" cluster. More formaly, if one denotes by $\text{Cluster}(T_i)$ the set of those clusters that lead to MCTs before the first jump for task T_i it is possible to define an order on the tasks as :

$$T_i \preccurlyeq T_j \Leftrightarrow \begin{cases} |\text{ Cluster}(T_i)\ | < |\text{ Cluster}(T_j)\ | \\ \text{or} \\ |\text{ Cluster}(T_i)\ | = |\text{ Cluster}(T_j)\ | \\ \quad \text{and } \text{Suff}(T_i) > \text{Suff}(T_j) \end{cases}$$

Sufferage II is then easily defined as always scheduling the minimum task according to order $\preccurlyeq$. That task, say T_i should be scheduled to a cluster in $clust(T_i)$ (e.g. the one that leads to the best MCT).