The AppLeS Parameter Sweep Template: User-level middleware for the Grid[1]

Henri Casanova[a], Graziano Obertelli[a], Francine Berman[a,*] and Richard Wolski[b]
[a]*Computer Science and Engineering Department, University of California, San Diego, LaJolla, CA 92093, USA*
E-mail: {casanova,graziano,berman}@cs.ucsd.edu
[b]*Computer Science Department, University of Tennessee, Knoxville, TN 37996, USA*
E-mail: rich@cs.utk.edu

The Computational Grid is a promising platform for the efficient execution of *parameter sweep applications* over large parameter spaces. To achieve performance on the Grid, such applications must be scheduled so that shared data files are strategically placed to maximize re-use, and so that the application execution can adapt to the deliverable performance potential of target heterogeneous, distributed and shared resources. Parameter sweep applications are an important class of applications and would greatly benefit from the development of *Grid middleware* that embeds a scheduler for performance and targets Grid resources transparently.

In this paper we describe a user-level Grid middleware project, the *AppLeS Parameter Sweep Template* (*APST*), that uses application-level scheduling techniques [1] and various Grid technologies to allow the efficient deployment of parameter sweep applications over the Grid. We discuss several possible scheduling algorithms and detail our software design. We then describe our current implementation of APST using systems like Globus [2], NetSolve [3] and the Network Weather Service [4], and present experimental results.

Keywords: Application level scheduling, adaptive runtime scheduling, computational grid, grid middleware, parameter sweeps applications, scheduling heuristics, distributed storage

1. Introduction

Fast networks make it possible to aggregate CPU, network and storage resources into *Computational Grids* [5,6]. Such environments can be used effectively to support large-scale runs of distributed applications. An ideal class of applications for the Grid is the class of *parameter sweep applications* (*PSAs*) that arise in many scientific and engineering contexts [7–13]. These applications are typically structured as sets of "experiments", each of which is executed with a distinct set of parameters. There are many technical challenges involved when deploying large-scale applications over a distributed computing environment. Although parameter sweep experiments are independent (i.e. do not communicate), many PSAs are structured so that distinct experiments share large input files, and produce large output files. To achieve efficiency for large-scale runs, shared data files must be co-located with experiments, and the PSA must be scheduled to adapt to the dynamically fluctuating delays and qualities of service of shared Grid resources. Previous work [1,14,15] has demonstrated that run-time, adaptive scheduling is a fundamental approach for achieving performance for such applications on the Grid.

PSAs are of great interest to the scientific community and user-level middleware targeted to the efficient execution and deployment of large-scale parameter sweeps would be enormously helpful for users. *In this paper we describe the design and implementation of such middleware: the AppLeS Parameter Sweep Template* (*APST*). The purpose of the template is to provide a framework for easily and efficiently developing, scheduling, and executing large-scale PSAs on Computational Grids.

Our work complements the Nimrod project [16] which also targets PSAs, as well as the work on developing high-throughput Condor Monte Carlo simulations [17]. Nimrod's scheduling approach is different from ours in that it is based on deadlines and on a Grid economy model, while our work focuses on the efficient co-location of data and experiments and adaptive scheduling. The Condor work also does not consider

*Corresponding author.
[1]This research was supported in part by NSF Grant ASC-9701333, NASA/NPACI Contract AD-435-5790, DARPA/ITO under contract #N66001-97-C-8531.

Scientific Programming 8 (2000) 111–126
ISSN 1058-9244 / $8.00

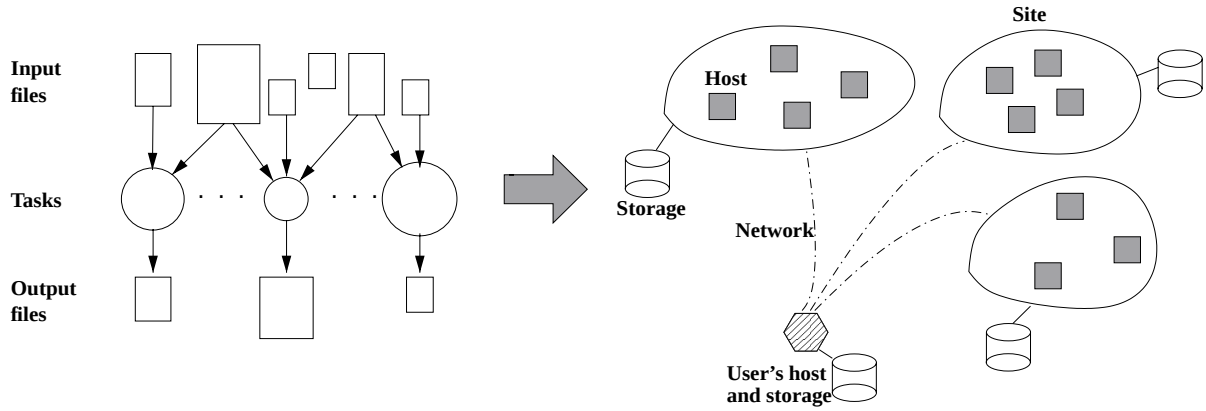

Fig. 1. Application and Grid model.

distributed data constraints. Moreover, our work differs from both Nimrod and Condor in that it is designed to target multiple Grid infrastructure environments simultaneously (following the example of EveryWare [18]).

In Section 2.1, we describe our application and Grid model. In Sections 2.2 and 2.3, we provide context by describing previous work on PSA scheduling algorithms. In Section 3, we describe new work on the design and implementation of the AppLeS Parameter Sweep Template. Section 4 presents experimental results and an evaluation of the APST software. We conclude with and discuss future work in Section 5.

2. Scheduling parameter-sweep applications

2.1. Application and Grid model

We define a *parameter sweep application* as a set of "independent" sequential tasks (i.e. with no task precedences). We assume that the input to each task is a set of files and that a single file might be input to more than one task. In our model, we assume without loss of generality that each task produces exactly one output file. This model is motivated by real-world PSAs (see Section 4.1).

We assume that the Computational Grid available to the application consists of network-accessible *sites* that contain computing resources called *hosts* (workstations, multi-processors, etc.), as well as local storage resources called *disks*. In the case of large-scale PSAs, it is critical for performance that hosts within a site can share data in local disks efficiently. Typically the number of computational tasks in the application will be orders of magnitude larger than the number of available processors.

The implementation of APST aims at leveraging whatever software infrastructure is available in the distributed environment. Access to remote computational resources can be facilitated by various Grid infrastructure projects [2,3,19–22], and several approaches can be used for implementing the distributed storage infrastructure (e.g. low-level systems such as GASS [23] and IBP [24], or distributed file systems such as AFS).

Figure 1 depicts both our application model and our Grid model. We do not impose any constraints on the performance characteristics of the resources. However, some of the scheduling algorithms described hereafter will require *estimates* of computation and file transfer times. Such estimates can be provided by the user, analytical models or historical information, by facilities such as the Network Weather Service (NWS) [4], ENV [25], Remos [26], and/or Grid services such as those found in Globus [2], or computed from a combination of the previous. Our models and assumptions are discussed in more detail in [27].

2.2. The self-scheduled workqueue

A straightforward and popular adaptive scheduling algorithm for scheduling sets of independent tasks is the *self-scheduled workqueue* [28] (*workqueue* for short). The algorithm assigns work to hosts as soon as they become available in a greedy fashion. Even though it is very adaptive, it may fail to capture many idiosyncrasies of the application with respect to the computing environment such as data storage requirements and data sharing patterns. In the context of our application model (see Fig. 1), a workqueue strategy is appropriate for certain application scenarios: if there are no large shared input files, or if large input files are shared by a very large number of tasks making file transfer

costs negligible compared to computational costs. The topology and nature of the Grid available to the user can also justify the use of a workqueue (e.g. large clusters interconnected with high-speed networks).

However, there are many instances in which the workqueue algorithm leads to poor schedules, typically when large files are shared by a relatively small number of tasks, or by many tasks that have relatively small computational costs. These situations arise in many real-world PSAs where the user explores ranges of parameters for multiple scenarios, each scenario being described in a large input file (see Section 4.3). Also, it may be that the relatively slow networks interconnecting the different sites combined with relatively large file sizes make it critical for the application that file transfer costs be minimized. This will certainly be true for future Grid-wide PSAs that make use of large datasets stored in distributed digital libraries over the Internet. Finally, and most importantly, the workqueue algorithm does not perform any resource selection on behalf of the application. Given the sheer amount of resources that will soon be available on the national Computational Grid, it will be necessary to determine subsets of resources that can be used effectively for running a PSA.

Thus, there is a need for a more sophisticated adaptive scheduling algorithm that can automatically perform on-the-fly resource selection and co-allocation of data and computation when needed. The following section describes our first approach at designing and implementing such an algorithm.

2.3. Adaptive scheduling with heuristics for task-host assignment

Efficient deployment of PSAs on the Computational Grid requires the use of scheduling algorithms specially designed for, and adapted to the structure and the requirements of PSAs. Given the dynamic nature of Grid environments, adaptive algorithms provide the flexibility to react to changing resource conditions at run-time. In a previous paper [27], we described a scheduling algorithm which provides a fundamental building block of the AppLeS Parameter Sweep Template. In this subsection, we review those results in order to make this paper self-contained.

For PSAs, we focus on scheduling algorithms whose objective is to minimize the application's makespan (as defined in [29]). In [27] we proposed an adaptive scheduling algorithm that we call `sched`(). The general strategy is that `sched`() takes into account resource performance estimates to generate a *plan* for assigning file transfers to network links and tasks to hosts. To account for the Grid's dynamic nature, our algorithm can be called repeatedly so that the schedule can be modified and refined. We denote the points in time at which `sched`() is called *scheduling events*, according to the terminology in [30]. Having multiple scheduling events makes it possible to have achieve adaptive scheduling. The more frequent the events, the more adaptive the algorithm.

Figure 2 shows the general skeleton for `sched`(). Step (1) takes care of setting the scheduling even intervals dynamically. In Step (2), `sched`() creates a Gantt chart [31] that will be used to build the scheduling plan. The chart contains one column for each network link and one for each host. Each column is a time line that can be filled with *blocks* to indicate resource usage. Step (3) inserts blocks corresponding to *ongoing* file transfers and computations. Step (4) is the core of the algorithm as it assigns tasks and file transfers to hosts and network links. Finally, in step (5), the Gantt chart is converted into a schedule that can be implemented on Grid resources.

The problem of deciding on an optimal assignment of file transfers to network links and computation to hosts is NP-complete. It is therefore usual to use heuristics to make those decisions and step (4) in the algorithm is where such heuristics can be implemented. Simple heuristics for scheduling independent tasks were proposed in [30,32]: *Min-min*, *Max-min*, and *Sufferage*. These heuristics iteratively assign tasks to processors by considering tasks not yet scheduled and computing expected Minimum Completion Times (MCTs). For each task, this is done by tentatively scheduling it to each resource, estimating the task's completion time, and computing the minimum completion time over all resources. For each task, a *metric* is computed using these MCTs, and the task with the "best" metric is assigned to the resource that lets it achieve its MCT. The process is then repeated until all tasks have been scheduled. The way of computing and evaluating the metric entirely defines the heuristic's behavior, and Appendix A gives the basic algorithm of the heuristics and succinct descriptions of four different metrics.

These *base* heuristics are effective in environments where tasks and hosts exhibit *affinities*, that is where some hosts are best for some tasks but not for others (e.g. due to specialized hardware, optimized software libraries, etc.). Our key idea here is that the presence of some input files in a disk "close" to some host is akin to the idea of task/host affinities. This is why we expect

```
sched() {
    (1) compute the next scheduling event
    (2) create a Gantt Chart, G
    (3) foreach computation and file transfer currently underway
            compute an estimate of its completion time
            fill in the corresponding blocks in G
    (4) until each host has been assigned enough work
            heuristically assign tasks to hosts (filling blocks in G)
    (5) convert G into a plan
}
```

Fig. 2. Scheduling algorithm skeleton.

the heuristics to be effective in our setting. We adapted all three base heuristics so that they take into account such file location constraints. Note that step (4) only fills the Gantt chart until "enough" work has been assigned to resources. Indeed, `sched()` will be called at the next scheduling event, and new assignments will then be made. Typically, step (4) assigns work to resources until the next scheduling event plus some fixed amount of time to be conservative (i.e. to account for prediction inaccuracies).

Intuitively, the rationale behind Sufferage seems to be most applicable to our setting: a task should be scheduled to a host when that task is the one that would "suffer" the most if not scheduled to that host. We expect the sufferage idea to be a simple, elegant, yet effective way to capture data distribution patterns. We proposed an extension to the Sufferage heuristic and adapted it to our Grid model to create the *XSufferage* heuristics (see Appendix A). All the heuristics mentioned here have been evaluated in simulation environments and XSufferage proved to be the most promising heuristic (see [27]).

We also conducted simulations to study the impact of inaccurate performance estimates as perfect predictive accuracy is rarely found in real programming environments. Increasing the frequency of the calls to the scheduling algorithm (in other words, increasing its adaptivity) makes it possible to tolerate these inaccuracies and simulations show that XSufferage performs particularly well in the context of multiple scheduling events and poor information. All the heuristics are designed to have low computational complexity. Furthermore, it is possible to reduce the set of tasks that are considered by the heuristics. Section 4.3 describes a first simple approach at such a *task space reduction*.

It is therefore possible to run `sched()` frequently. Note that a self-scheduled workqueue algorithm does not make use of any performance estimates. In the case of a single scheduling event (at the beginning of execution), the workqueue is the most performance-efficient scheduling algorithm when the accuracy of performance estimates is poor.

3. The AppLeS Parameter-Sweep Template

3.1. Motivation and goals

The AppLeS project [1,33] focuses on the design and development of Grid-enabled high-performance schedulers for distributed applications. The first generation of AppLeS schedulers [34] demonstrated that simultaneously taking into account application- and system-level information makes it possible to effectively schedule applications onto computational environments as heterogeneous and dynamic as the Grid. However, each scheduler was embedded within the application itself and thus difficult to re-use for other applications. The next logical step was to consider *classes* of applications that are structurally similar and to develop independent software frameworks, i.e. *templates*, that can schedule applications within a class. This paper focuses on the AppLeS Parameter-Sweep Template (APST) which targets PSAs.

Our goal is twofold. First, we seek to use APST to investigate the difficult problems of adaptive scheduling and deployment of PSAs. Second, we seek to provide users with a convenient and efficient way of running PSAs over most available Grid resources. APST and projects like SciRun [35] and Nimrod/G [16] provide initial examples of user-level Grid middleware. The development of such user-level middleware will be critical to the wide-spread use of and application performance on the Computational Grid.

To achieve both our goals, we must design the software so that it is possible for the user to enable/disable different features (e.g. using Globus resources or not), and tune various parameters of the scheduling algo-

rithm (e.g. choosing a scheduling heuristic). Ultimately, most tuning should occur automatically as described in Section 5. The following section describes and justifies our design choices.

3.2. Software design

Figure 3 shows the overall design of the AppLeS Parameter Sweep Template. The software is composed of a *client* and a *daemon*. At the moment, the client is an executable that takes various command-line arguments and can be used by the user to interact with the daemon. It is possible to use the client to submit new computational tasks, cancel tasks previously submitted, and inquire about the status of an ongoing computation. To submit new tasks, the user must provide a *task description file* which contains one task description per line. Each task description specifies which program to run as well as the required command-line arguments, the location of input files, and where output files should be created (i.e. written to local disks or left in place in remote storage). The user also has the option to provide an estimate of the task's relative computational cost. We focus on the scheduling algorithm rather than on developing user interfaces. However, we expect that the APST client in its current form can serve as a building block for more sophisticated user interfaces (e.g. within a PSE such as SCIRun [35] or Nimrod [16], or from a Web/CGI interface). Current APST users generally use the client from the shell and generate task description files with simple Perl scripts. The current implementation of the daemon assumes a single client (i.e. user) at the moment.

As seen in Fig. 3 the APST daemon consists of four distinct sub-systems: the Controller, the Scheduler, the Actuator, and the Meta-data Bookkeeper. Each subsystem defines its own API. Those APIs are used for communication/notification among sub-systems. Providing multiple implementations of these APIs makes it possible to *plug in* different functionalities and algorithms into each APST sub-system. For instance, writing multiple implementations of the Scheduler's API is the way to provide multiple scheduling algorithms.

The *Scheduler* is the central component of the APST daemon. Its API (*sched_api*) is used for notification of events concerning the application's structure (new tasks, task cancellations), the status of computational resources (new disk, new host, host/disk/network failures), and the status of running tasks (task completions or failures). The behavior of the scheduler is entirely defined by the implementation of this API. The *Controller* relays information between the client and the daemon and notifies the Scheduler of new tasks to perform or of task cancellations. It uses the Scheduler's API and communicates with the client using a simple wire protocol. The *Actuator* implements all interaction with Grid infrastructure software for accessing storage, network, and computation resources. It also interacts with the Grid security infrastructure on behalf of the user when needed. There are two parts to the Actuator's API: (i) the *transport_api* handles file transfer and storage; (ii) the *env_api* handles task launching, polling, and cancellation. The Scheduler can place calls to both these APIs to have the Actuator implement a given schedule on Grid resources. The Actuator's implementation makes use of standard APIs to Grid infrastructure softwares to interact with resources. Each sub-system API consists of less than 15 functions.

Our design ensures that it is possible to mix and match different implementations of the APIs. In particular, the implementation of a given scheduling algorithm is completely isolated from the actual Grid software used to deploy the application's tasks. Section 3.3 describes the implementations that are currently available.

The intent of our design is that a constant control cycle between the Scheduler and the Actuator is necessary for effective scheduling. Using the env_api, the Scheduler periodically polls the Actuator for task completions, failures, or other events (e.g. newly available computing resources). This leads the Actuator to place calls to the sched_api to notify the Scheduler of these events. The Scheduler has then the opportunity to react by making decisions and placing calls to the env_api and the transport_api. Such a design makes it very easy to implement straightforward algorithms like a self-scheduled workqueue, as well as more complex algorithms as the one presented in Section 2.3.

As seen in Section 2.3, promising scheduling algorithms base their decisions in part on *forecasted* computation and file transfer times. The *Meta-data Bookkeeper* is in charge of keeping track of static and dynamic meta-data concerning both the resources and the application. It is also responsible for performing or obtaining forecasts for various types of meta-data. Its API (*meta_api*) contains two functions: one to store meta-data local to the application inside the Bookkeeper; the other to obtain a forecast for (dynamic) meta-data. Dynamic and static meta-data concerning Grid resources are available via Grid Information Services (GISs) and accessible via standard Grid APIs. In addition, the Actuator stores meta-data for the application (e.g. ob-

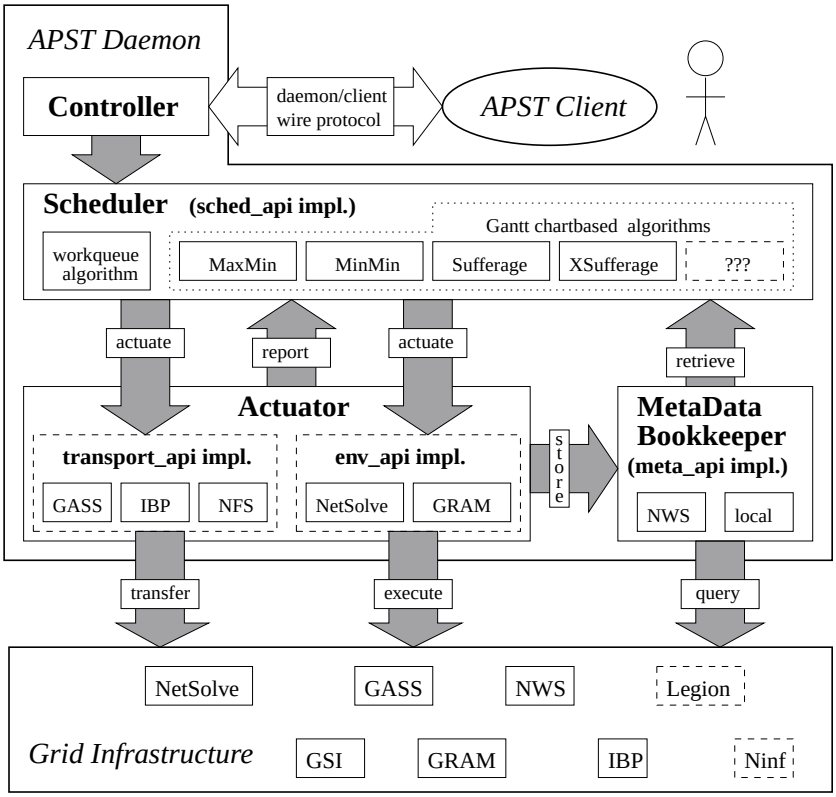

Fig. 3. The AppLeS APST software design.

served file transfer duration on some given network link) inside the Bookkeeper. The Scheduler requests forecasts for dynamic meta-data on which it can base scheduling decisions. The following section gives more details on the implementation of the forecasting facility.

3.3. Current implementation

A beta prototype implementation of APST was demonstrated during the Supercomputing'99 conference (running over 200 hosts). This document describes APST v1.0. The software has been developed on Linux and ported to most UNIX platforms. Let us review the implementation of each of the sub-systems presented in the previous section.

We have implemented multiple versions of the sched_api for each of the following scheduling algorithms: a standard workqueue algorithm, a workqueue algorithm with work-stealing and task-duplication, and a Gantt chart algorithm that can use any of the heuristics introduced in Section 2.3. At the moment, the algorithm and its parameters are chosen when starting the APST daemon, and stays effective for the life-cycle of the daemon. As described in Section 5, we plan to perform further investigation to allow for automatic and dynamic scheduling algorithm selection without user intervention.

The Bookkeeper interacts with the Network Weather Service (NWS) [4,36] for obtaining dynamic Grid resource information concerning CPU loads, as well as network latencies and bandwidths. Forecasting is done using the NWS forecasting module directly linked in with the APST daemon (we use NWS version 2.0). The alternative would be to query remote NWS forecasters for NWS-generated meta-data, and still use the linked-in NWS forecaster for APST-generated meta-data. The use of remote forecasters would minimize forecast-related network traffic as potentially large time series need not be transmitted over the network. On the other hand, the use of a linked-in NWS forecaster makes the implementation simpler as it does not require a distinction between the two types of meta-data. We will make a final decision on that trade-off once we gain more experience with the current implementation.

The Actuator handles all interaction with Grid software for moving files and launching jobs. Currently, we provide three implementations of the transport_api. Two are on top of Grid storage software: GASS [23]

which is part of the Globus toolkit, and the Internet Backplane Protocol (IBP) [24] developed at the University of Tennessee. Both these projects provide some level of abstraction for handling remote/distributed storage devices. The third implementation of the transport_api is on top of NFS. It can be used when a host used for computation can directly access the user's file system. We currently have 2 implementations of the env_api: one on top of Globus' GRAM [37], and one on top of NetSolve [3]. NetSolve is a client-agent-server system that offers a simple way to submit computations to remote servers, and in its current version it does not implement any security mechanisms. On the other hand, the GRAM implementation of the env_api makes use of the Globus Security Infrastructure (GSI) to submit jobs. We are also considering providing implementations of the env_api for Ninf [21] and Legion [19].

The Actuator's design provides great flexibility as it is possible to mix and match the env_api and the transport_api. The APST daemon can then simultaneously use Globus and NetSolve servers for tasks, as well as IBP, GASS, and NFS servers for storage. It is also possible to have any task (spawned by the the env_api) use files in any storage system. At the moment however, our implementation imposes the restriction that GRAM-spawned tasks can only access files in GASS servers or over the NFS. Likewise, NetSolve-spawned tasks can only access files stored in IBP servers or over the NFS. Using systems such as GASS and IBP makes it possible for tasks outside the client's file system to *share* copies of a single file as described in Section 2.1. This in turn makes it worthwhile to use the heuristics of Section 2.3.

4. Software evaluation

4.1. The MCell PSA

Parameter Sweep Applications arise in various fields [8–13,38]. In this work we focus primarily on MCell [7,39], a micro-physiology application that uses 3-D Monte-Carlo simulation techniques to study molecular bio-chemical interactions within living cells. MCell can be used to study the trajectories of neurotransmitters in the 3-D space between two cell membranes for different deformations of the membranes. MCell is currently being used in over 20 laboratories over the world for practical applications. It is typical for subsets of MCell tasks to share large data files (e.g. describing 3-D polygon surfaces) and our expectation is that scheduling heuristics such as XSufferage will lead to good performance in real-world Grid environments.

We use three criteria to evaluate our software with MCell. First, we discuss how APST's *usability* makes it not only possible but easy for users to deploy large scale MCell simulations. Second, we use experimental results to show how APST's *scheduling* capabilities promote performance in a Grid environment. Third, we show how the use of *multi-threading* allows the APST daemon to achieve better Grid resource utilization.

4.2. Usability

Previously, MCell users wanting to run a distributed MCell simulation had to manually upload input files to remote computational sites, create a collection of Shell scripts to perform simulation sequentially on each available host, manually check for task completions and download produced output file. There was no support for data-management, scheduling, or fault-tolerance. APST addresses all three issues. The user needs to simply write task descriptions (see Section 3.2) and provide paths to input files on his/her local disk or to files that have been pre-staged on remote storage. The user never performs any explicit data upload and download as those are determined by the Scheduler and transparently implemented via Grid services. Scheduling decisions are made automatically and are based on real-time resource availabilities and loads. Finally, APST inherits fault-tolerance mechanisms from Grid computational services and can recover from host failures gracefully. In addition, new resources can be added (or deleted) on-the-fly and APST adapts its current schedule accordingly. MCell users can now deploy their simulations over large sets of resources without modifying the way they design their experiments. As far as the user is concerned, the simulation runs locally on the desktop.

4.3. Scheduling algorithm

Figure 4 depicts the computing environment that we used for the experiments reported in this section. Hosts were located in three different institutions: the University of California at San Diego (UCSD), the University of Tennessee, Knoxville (UTK), and the Tokyo Institute of Technology (TITECH). Hosts at UCSD and TITECH are Linux/Intel boxes, whereas hosts at UTK are Solaris/Sparc workstations. We deployed NetSolve and IBP at TITECH and UTK, and the Globus's GRAM and GASS at UCSD (version 1.1.3). The host break-

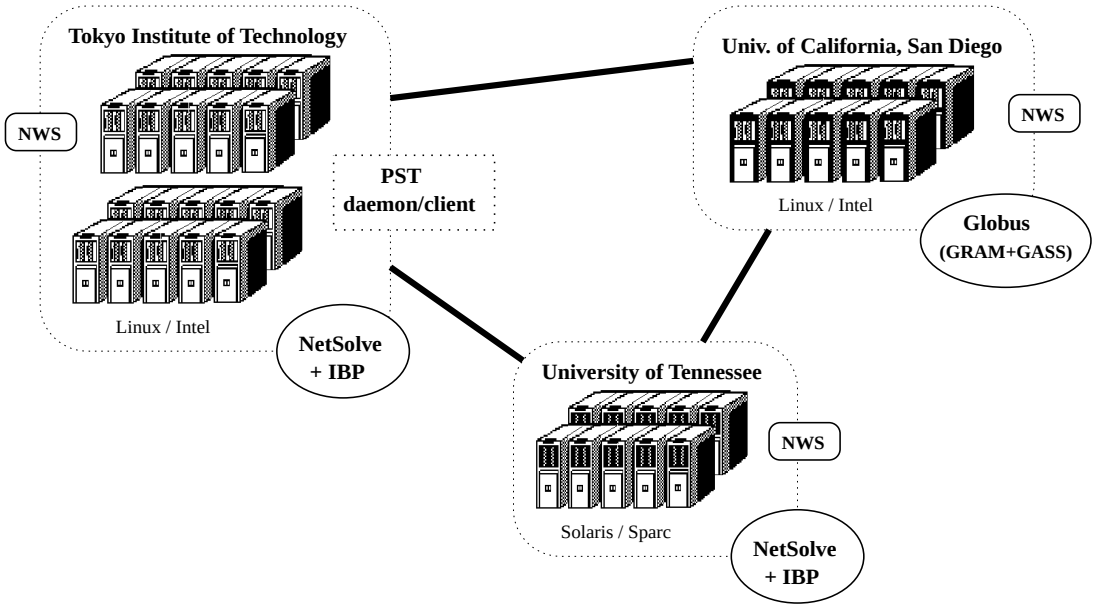

Fig. 4. Experimental Grid configuration.

down for the three institutions is as follows: 8 hosts at UCSD, 16 hosts at UTK, and 32 hosts at TITECH. All hosts were used in non-dedicated mode and the NWS was deployed to monitor CPUs and network links. We had to start the APST daemon and client within the TITECH firewall so that we could make use of NetSolve to access hosts on the TITECH cluster. Note that current developments (e.g. [40]) will make it possible to use hosts behind firewalls in future experiments.

4.3.1. Experiments

The first set of experiments was conducted with a standard MCell application consisting of 1,200 tasks. Relative task computational costs are in the range 1–20 (meaning that the most expensive task required 20 times more computation than the cheapest one). The application is structured as 6 Monte-Carlo simulations (each containing 200 tasks) and all tasks within a simulation share a common input file describing the geometry of a different 3-D space. The 6 shared files are of size 1, 1, 20, 20, 100 and 100 MBytes respectively. Other input files and produced output files are all under 10 Kbytes. All input files are initially present in local storage on the host running the APST client, that is a host at TITECH that we will call the *source*.

In these experiments, we compare two different schedulers: the self-scheduled workqueue and the Gantt chart algorithm introduced in Section 2.3. Figure 5 shows results obtained with 4 different scenarios. On that figure, bar heights are averages over 5 runs, and two standard deviations are shown as error bars. For each scenario, measurements were obtained for back-to-back runs of the MCell simulation with both schedulers, and all experiments were performed over a period of 4 days. In all scenarios all input files are available from the source and in all but scenario (a) some shared input files have been *pre-staged* at some remote sites. This pre-staging can be actively performed by the user while setting up the MCell simulation, can be the result of using distributed digital libraries that use replication for better efficiency, or can just mean that left-over files from previous runs can be re-used. The extreme scenario (d) is when all shared input files are pre-staged at all remote sites. The expectation is that the Gantt chart algorithm will outperform the workqueue when few files have been pre-staged, whereas the performance of both algorithms should be similar when many files have been pre-staged, making the Gantt chart algorithm the most versatile and consistent overall.

Note that we do not give results for all the heuristics mentioned in Section 2.3, but instead just give generic "Gantt chart" results. In fact, we did not observe any real difference between the heuristics in these experiments as their scheduling choices were similar. This is due to the topology of our testbed in Fig. 4. Indeed, the simulation results previously obtained in [27] indicated that heuristics start behaving differently when the Grid topology contains a larger number of sites. Although time constraints prevented us from gathering experi-

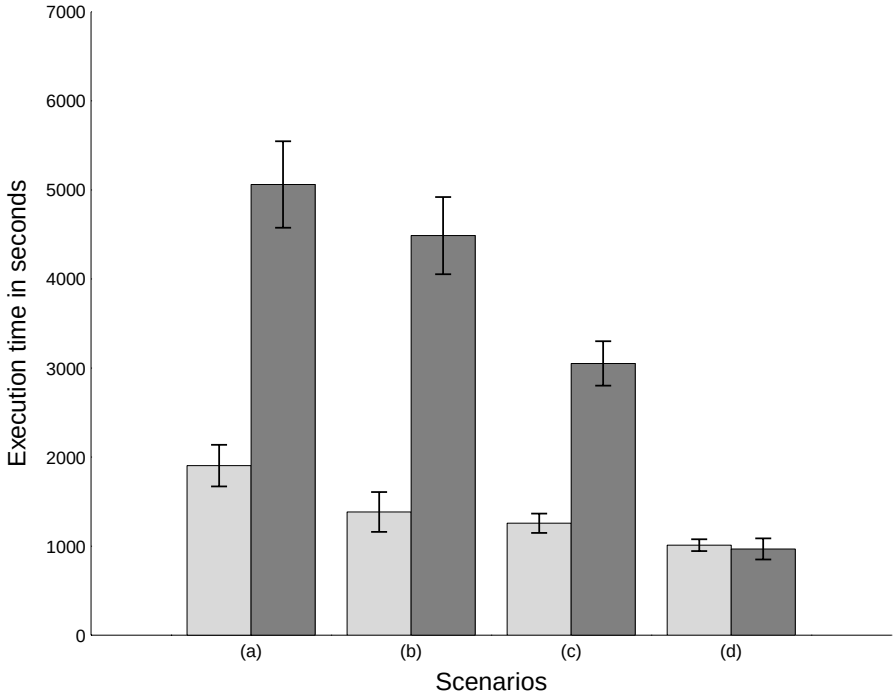

Fig. 5. Gantt chart vs. workqueue in 4 scenarios.

mental results on a larger testbed here, we will report on more extensive experiments in a future paper.

On average, during the experiments, the bandwidths between the source and the TITECH cluster is 20 times higher than the one between the source and the UCSD cluster, and 40 times higher than the one between the source and the UTK cluster. In scenario (a) no file are pre-staged. The workqueue sends tasks to hosts in a greedy fashion, leading to large input files transfers over slow links. By contrast, the Gantt chart algorithm uses the UTK cluster only for tasks that use small input files (1 Mbyte), the UCSD cluster for tasks that use small or medium input files (20 MBytes), and the TITECH cluster for all tasks. This explains the average 62% gap between the two scheduling algorithms. In scenario (b), the 100 MByte input files have been pre-staged at UCSD. Expectedly, both scheduling algorithms show improvement, but the workqueue still pays the cost of sending 100 MByte files over to UTK. In scenario (c), one 100 MByte and the two 20 MByte input files have been pre-staged on the cluster at UTK, in addition to the files already pre-staged for scenario (b). Again, this leads to improvements for both algorithms by comparison with scenario (b). The relative improvement if larger for the workqueue as one of the 100 MByte file needs not be transfered. Finally, in scenario (d), all large input files are pre-staged on all clusters, and one can see that both workqueue and the Gantt chart algorithm lead roughly to the same performance. These results indicate that using the scheduling algorithm of Fig. 2 leads to better schedules because it takes into account data storage and data sharing patterns in the application. When there is no data storage concern (as in scenario (d)), then it performs similarly to a workqueue algorithm.

4.3.2. Forecast accuracy and scheduling cost

Given the large number of available resources and of tasks in PSAs, we have to use techniques to minimize the time required to apply the heuristics of Section 2.3. An obvious cost incurred by the heuristics is the one of obtaining forecasts from the Meta-Data Bookkeeper. Indeed, as detailed in Appendix A, each heuristics makes heavy use of performance estimates, and constantly obtaining new forecasts leads to prohibitive costs. Therefore, our implementation of the Bookkeeper provides *caching* of forecasts that can be re-used until a new forecast is obtained. This caching mechanism is transparent to the other APST components. Figure 6 shows typical percentage relative errors when forecasting task execution times during an MCell run. We mentioned in Section 3.2 that the user can provide relative computational costs for the application

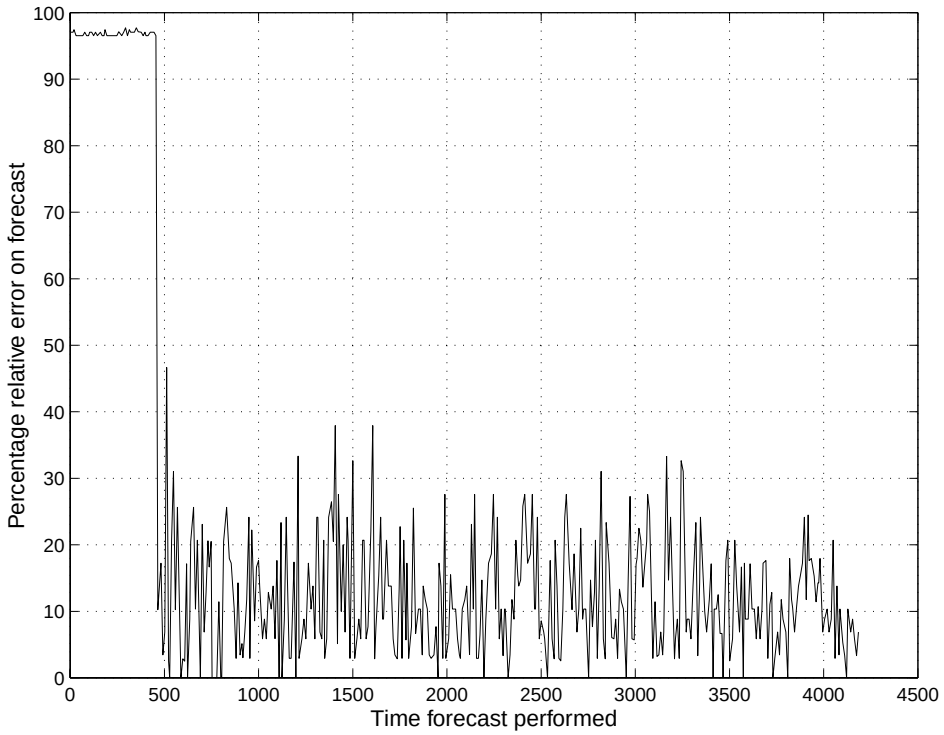

Fig. 6. Forecast accuracy.

tasks. These costs are not representative of real task execution times, but can be used to compare tasks (e.g. a task with cost "2" requires twice as much computation as a task with cost "1"). During the first round of scheduling, the Bookkeeper makes initial guesses for actual task execution times. Once tasks complete, the Bookkeeper uses observed execution times to improve accuracy by factoring in the user-provided relative costs, observed CPU loads, and host speeds (when unloaded). Figure 6 shows that behavior, with relative errors initially around 100%, and then dropping to an average of 11% after about 500 seconds which is when the first tasks complete. In this experiment scheduling events where 250 seconds apart, and according to the simulation result obtained in [27] a relative error rate of 11% is largely sufficient to justify the use of Gantt chart heuristics for scheduling. This corroborates the results in the previous section.

As seen in Appendix A, the heuristics are implemented with nested loops over all tasks that have not yet been scheduled. When the total number of tasks in the application is large, the heuristics can become costly (albeit in polynomial time). It is then necessary to run the heuristic on a *reduced* task space. The current implementation of the heuristics in the APST Scheduler reduces the task space by limiting the number of tasks processed by the inner loop of each heuristic's algorithm to a fixed number of tasks (200 in our experiment). These 200 tasks are randomly selected at each iteration of the outer loop. For comparison purposes we ran a few experiments without task-reduction, and did not observe any degradation in the performance of the obtained schedules. In fact, as long as the set of 200 tasks contains one task that uses each large shared input file, the heuristic makes the right decisions. Statistically, this is almost always the case for the MCell application we used as there are only 6 distinct large input files. This will not be true for applications that exhibit more complex structures, and more sophisticated task space reduction technique will be needed. More detailed investigations on how different reduction techniques impact the quality of the resulting schedule is left for future work.

Due to the caching mechanisms used by the Bookkeeper and our simple task space reduction, the time required to run the `sched`() algorithm averaged 10 seconds for all runs (recall that `sched`() was called every 250 seconds). Further optimization of the Gantt chart implementations are possible, and the released version of APST will likely exhibit even lower scheduling costs.

4.4. Multi-threading for performance

It is critical that the Actuator embedded in the APST daemon be able to launch computations on Grid resources (in our case NetSolve and Globus resources) as efficiently as possible. Indeed, since PSAs consist of many tasks and are to be deployed over many resources, the overhead incurred when starting remote computations may have dramatic impact on the overall execution time. This overhead is due to both network and software latencies. For instance, to launch a computation on a remote NetSolve server, the NetSolve client (transparently to the user) contacts a NetSolve agent, downloads an IDL description of a few Kbytes, and performs an IDL check with the remote server. The server then forks a separate process to handle the computation, and that process sends an acknowledgement back to the client. The NetSolve client then returns control to the APST Actuator. Similar overheads are incurred when launching remote jobs via the Globus GRAM.

Let us give an example: say that the APST daemon has access to 60 identical remote hosts for computation, and that the MCell simulation to be performed consists of, 1000 tasks that all require 30 seconds of computation. For the sake of discussion, assume that the Actuator incurs and overhead of 1 second for initiating a computation on a remote host. In this setting, under the optimistic assumption that there is no overhead for acknowledging task completions, the APST daemon will achieve at most 50% utilization of available resources, leading to an execution time roughly two times larger than the execution time without the 1 second overhead. APST users trying to run large MCell computations faced that problem acutely when performing simulations on the SP-2 available at the San Diego Supercomputing Center that provides more than 1000 nodes for computation. The overhead for starting a remote job was on average 0.3 seconds and the machine utilization for typical MCell simulation, at times, got as low as 20%.

Our first step was to dynamically measure and include that overhead into the Scheduler's performance model (e.g. when filling the Gantt chart). While improving the schedule's accuracy, the new performance model led to no observable improvement from the user's perspective. We then opted for providing a more efficient implementation of the env_api, and our first approach uses threads. The expectation is that multiple concurrent threads can place simultaneous call to Grid software, and that network and software latencies can be partially hidden. Some minor changes to the NetSolve client library were required in order to make it thread-safe whereas Globus provides a thread-safe GRAM API.

For experiments in this section we used NetSolve on all nodes of the "Presto" cluster, a 64-node 350MHz Pentium II Linux cluster made available to us by the Tokyo Institute of Technology (TITECH). More information on that cluster is available at [41]. As for the experiments in Section 4.3.1, the APST daemon was running also at TITECH, inside the firewall, but not on the cluster itself. We conducted experiments with an MCell simulation consisting of 1000 tasks. During the experiments, MCell tasks were running for 30 to 80 seconds (depending on the task and the CPU loads, since the cluster was not dedicated). Input/output file transfers were done with IBP, and we used the standard workqueue scheduler (which is as good a choice as any other in such an environment).

Figure 7 shows execution times for increasing number of *allowed* simultaneous threads in the env_api. For each number of allowed threads (1, 2, 3, 4, 10, 20 and 30), we show data points for 5 runs, for a total of 35 runs of the application. The execution time averages are linked with a solid line. One can see that the use of multi-threading leads to dramatic improvement and effectively hides network and software latencies. Allowing for 20 concurrent threads improves the execution time by 48% over no multi-threading. However, allowing for more than 20 threads does not lead to further improvement. This behavior is explained by the data in Fig. 8. On that figure, for the same experiments, we plot average numbers of simultaneous threads that the Actuator could effectively spawn. As in Fig. 7, we plot a data point for each run and link the averages with a solid line. One can see that, for this experiment, the Actuator never had the opportunity to spawn more than 12 threads. This is strongly related to the rate at which tasks complete in this particular setting, which in turns depends on the nature of the application and on the number and nature of available resources. In this experiment, the APST daemon spawns new tasks approximately every 10 seconds, in which time 12 tasks complete on average.

Finally, Fig. 9 shows the decrease in latency for individual NetSolve calls when the number of concurrent threads increases. The per-call latency is computed by dividing the time spent in spawning n NetSolve calls (in a multi-threaded fashion) by n. The data-points shown on Fig. 9 are averages computed over all 35 experiments previously reported in Fig. 7. The curve shows

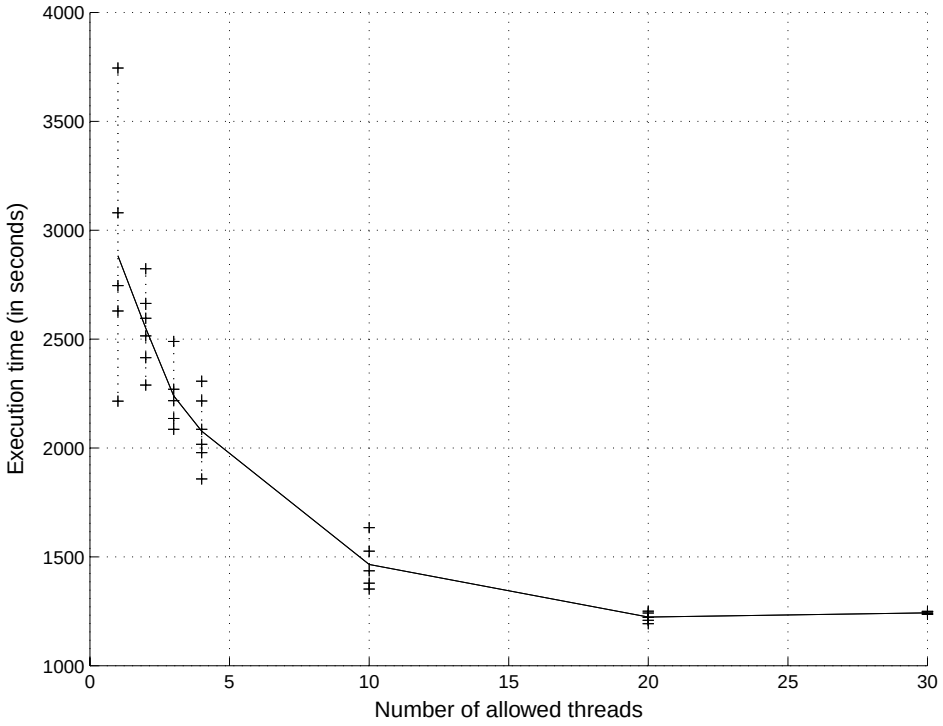

Fig. 7. Multi-threading and execution time.

a dramatic drop in per-call latency initially and then stabilizes at roughly 15 threads. This means that, when using NetSolve on this cluster, multi-threading is effective for hiding latency effectively only up to 15 threads. We expect that larger numbers of threads can be effective for larger network latencies and we will perform more experiments with different configurations. Since the APST daemon spawns 12 threads simultaneously on average in this experiment, it is close to making best use of multi-threading.

Following the same ideas, we implemented a multi-threaded implementation of the transport_api that allows concurrent file transfers on top of GASS, IBP and NFS. Threads could also be used to hide latencies associated with overheads incurred when acknowledging task completions. We leave this for a future implementation of the APST Actuator.

5. Conclusion and future work

In this paper we have described a user-level Grid middleware project, the AppLeS Parameter Sweep Template (APST), that is used to deploy Parameter Sweep Applications (PSAs) across the Computational Grid. The design of the software makes it easy to: (i) efficiently and adaptively use resources managed by multiple Grid infrastructure environments for large-scale PSAs; and (ii) investigate the effectiveness of a variety of adaptive scheduling algorithms and implementation strategies.

Our work on scheduling can be extended in several ways. We plan to integrate and possibly adapt more scheduling algorithms found in the literature such as [42,43]. Our current Grid model restricts network communications between the user's machine and the remote site; extending it to allow inter-site communication is straightforward and will be our next undertaking. All the scheduling algorithms mentioned in this paper are adaptive to changing resource conditions. However, our overall goal is to have APST select which algorithm should be used at run-time. In the release version of APST, we plan to dynamically select a workqueue algorithm or XSufferage depending on estimation accuracy. We have also initiated a collaboration with the Nimrod team in a attempt to explore Grid economy models and their impact on scheduling algorithms. We will investigate in more details the impact of several of the techniques we used to reduce the cost of computing a schedule, particularly task space reduction techniques and NWS forecast caching. Finally, we will perform many more experiments with larger testbeds and appli-

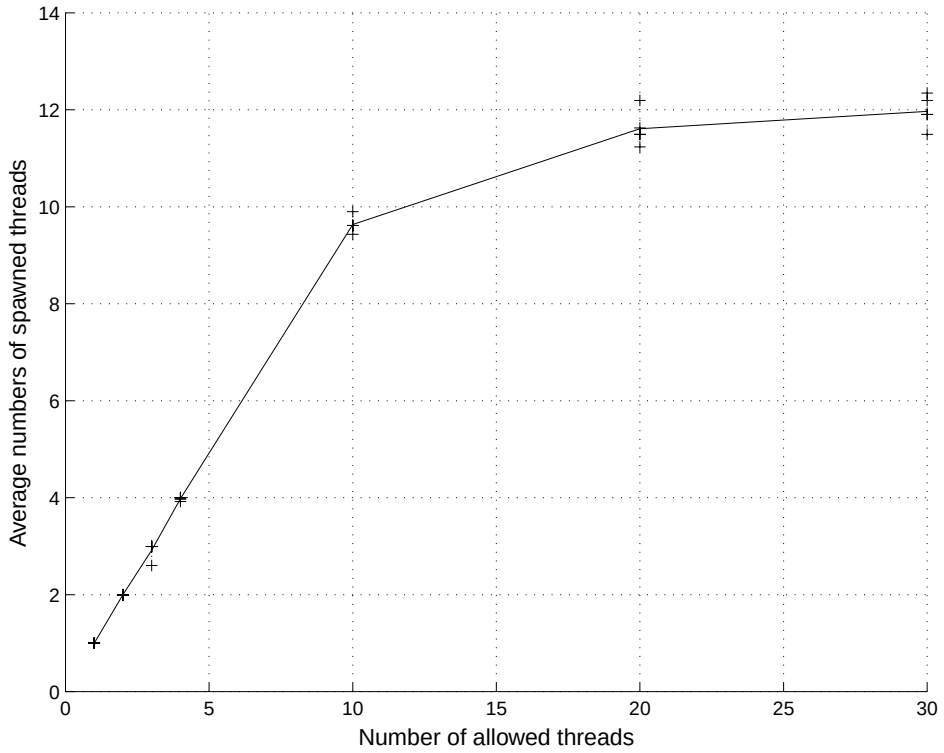

Fig. 8. Opportunities for multi-threading.

cations. We expect that these experiments will make it possible to effectively compare the different heuristics for assigning tasks to hosts.

From the software standpoint, several improvements can be made to the current implementation of APST. First, we must allow the APST daemon and the APST client to reside on different file systems. This will allow the same daemon to be used by multiple users for different applications. Multiple applications will require finding a way for the scheduling algorithms to ensure some degree of fairness among users without compromising efficient use of storage and computation resources. Our work motivates the development of better long-range forecasting techniques as part of the NWS, and the authors are collaborating with the NWS team towards that purpose. More components of the Globus toolkit need to be integrated/used by APST (e.g. HBM, MDS). We will also develop implementations of the APST Actuator over other Grid softwares like Ninf [21] and Legion [19]. Finally, following the idea presented in Section 4.4, we will push the use of threads further in the implementation of the Actuator for increased efficiency.

As Grid services become more available and ubiquitous, we plan to deploy large-scale parameter sweep simulations in production mode on Grid resources. These simulations will be executed at a scale that will enable users to achieve new disciplinary results.

Acknowledgements

The authors would like to thank the reviewers for their insightful comments, the Tokyo Institute of Technology for providing access to many computational resources, the MCell team for coping with us computer scientists, and members of the AppLeS group for their comments and help with the experiments.

Appendix A: Task/host selection heuristics

The general algorithm for the heuristics introduced in Section 2.3 is as follows (CT denotes the completion time):

while there are tasks to schedule
  foreach task i to schedule
    foreach host j
      compute $CT_{i,j}$ = CT(task i, host j)

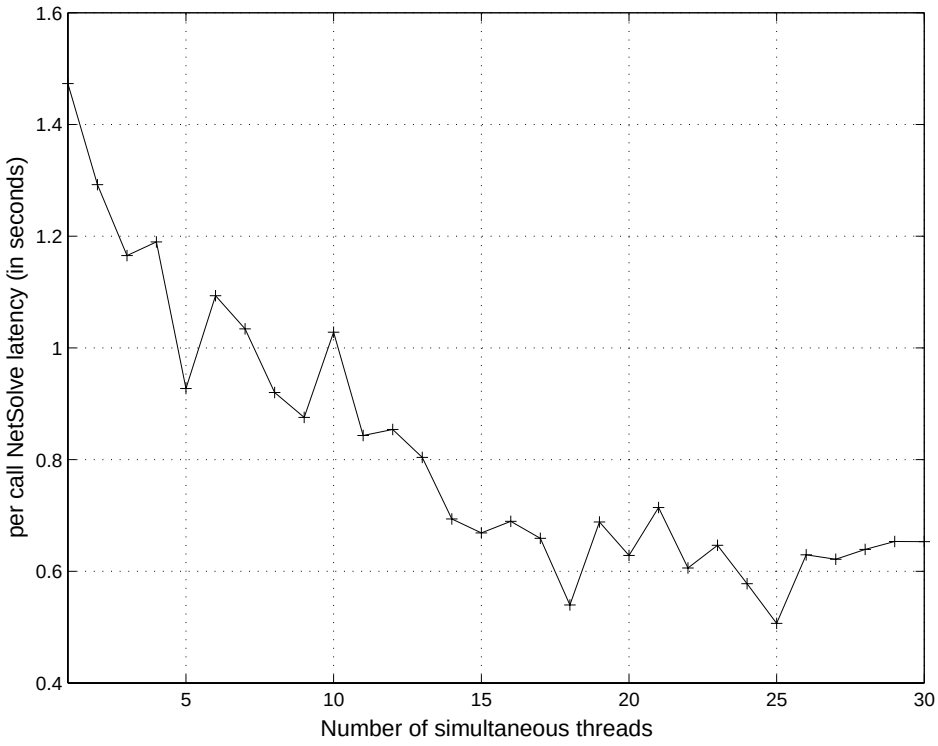

Fig. 9. NetSolve latency and multi-threading.

end foreach
compute $metric_i = f(CT_{i,1}, CT_{i,2}, ...)$
end foreach
choose "best" $metric_{i'}$
compute minimum $CT_{i',j'}$
schedule task i' on host j'
end while

Defining the function f and the meaning of "best" in this algorithm entirely defines a heuristic. We list here what these definitions are for our 4 heuristics.

MinMin: f is the minimum of all the $CT_{i,j}$. "Best" is defined as the minimum.

MinMin: f is the minimum of all the $CT_{i,j}$. "Best" is defined as the maximum.

Sufferage: f is the difference between the second minimum $CT_{i,j}$ and the minimum $CT_{i,j}$. This difference is called the *sufferage value*. "Best" is defined as the maximum.

XSufferage: For each task and each site f computes the minimum completion times of the task over the hosts in the site. We call this minimum the *site-level completion time*. For each task, f returns the difference between the second minimum and the minimum site-level completion time. We call that difference the *site-level sufferage value*. "Best" is defined as the maximum.

References

[1] F. Berman, R. Wolski, S. Figueira, J. Schopf and G. Shao, Application-Level Scheduling on Distributed Heterogeneous Networks, *Proc. of Supercomputing'96, Pittsburgh*, 1997.
[2] I. Foster and K. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications* **11**(2) (1997), 115–128.
[3] 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).
[4] R. Wolski, Dynamically Forecasting Network Performance Using the Network Weather Service, 6th High-Performance Distributed Computing Conference, August 1997, pp. 316–325.
[5] I. Foster and C. Kesselman, eds, CHAPTER= 12, *The Grid, Blueprint for a New computing Infrastructure*, (Chapter 12), Morgan Kaufmann Publishers, Inc., 1998.
[6] http://www.gridforum.org.
[7] 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* (1998), 279–284.

[8] D. Abramson, M. Cope and R. McKenzie, Modeling Photochemical Pollution using Parallel and Distributed Computing Platforms, in: *Proceedings of PARLE-94,* 1994, pp. 478–489.

[9] J. Basney, M. Livny and P. Mazzanti, Harnessing the Capacity of Computational Grids for High Energy Physics, in: *Conference on Computing in High Energy and Nuclear Physics,* 2000.

[10] W.R. Nelson, H. Hirayama and D.W.O. Rogers, The EGS4 Code system, Technical Report SLAC-265, Stanford Linear Accelerator Center, 1985.

[11] S.J. Sciutto, AIRES users guide and reference manual, version 2.0.0, Technical Report GAP-99-020, Auger project, 1999.

[12] A. Amsden, J. Ramshaw, P. O'Rourke and J. Dukiwica, Kiva: A computer program for 2- and 3-dimensional fluid flows with chemical reactions and fuel sprays, Technical Report LA-10245-MS, Los Alamos National Laboratory, 1985.

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

[14] N. Spring and R. Wolski, Application Level Scheduling: Gene Sequence Library Comparison, in: *Procedings of ACM International Conference on Supercomputing 1998,* July 1998.

[15] F. Berman, *The Grid, Blueprint for a New computing Infrastructure,* (Chapter 12), I. Foster and C. Kesselman, eds, Morgan Kaufmann Publishers, Inc., 1998.

[16] D. Abramson, J. Giddy, I. Foster and L. Kotler, High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? in: *Proceedings of the International Parallel and Distributed Processing Symposium,* May 2000, to appear.

[17] J. Basney, R. Raman and M. Livny, High-throughput Monte Carlo, in: *Proceedings ot the Ninth SIAM Conference on Parallel Processing for Scientific Computing,* March 1999.

[18] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring and A. Su, Running EveryWare on the Computational Grid, *Proceedings of Supercomputing 1999,* November 1999.

[19] A. Grimshaw, A. Ferrari, F.C. Knabe and M. Humphrey, Wide-Area Computing: Resource Sharing on a Large Scale, *IEEE Computer* **32**(5) (May 1999), 29–37.

[20] 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,* Department of Computer Science, University of Winsconsin, Madison, June 1988, pp. 104–111.

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

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

[23] 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.

[24] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany and R. Wolski, The Internet Backplane Protocol: Storage in the Network, in: *Proceedings of NetSore'99: Network Storage Symposium, Internet2,* 1999.

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

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

[27] H. Casanova, A. Legrand, D. Zagorodnov and F. Berman, Heuristics for Scheduling Parameter Sweep Applications in Grid Environments, in: *Proceedings of the 9th Heterogeneous Computing Workshop* (*HCW'00*), May 2000, pp. 349–363.

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

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

[30] 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*), April 1999.

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

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

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

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

[35] S. Parker, M. Miller, C. Hansen and C. Johnson, An integrated problem solving environment: The SCIRun computational steering system, in: *Proceedings of the 31st Hawaii International Conference on System Sciences* (*HICSS-31*), (Vol. VII), January 1998, pp. 147–156.

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

[37] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith and S. Tuecke, A Resource Management Architecture for Metacomputing Systems, in: *Proceedings of IPPS/SPDP'98 Workshop on Job Scheduling Strategies for Parallel Processing,* 1998.

[38] A. Majumdar, Parallel Performance Study of Monte-Carlo Photon Transport Code on Shared-, Distributed-, and Distributed-Shared-Memory Architectures, in: *Proceedins of the 14th Parallel and Distributed Proceeding Symposium, IPDPS'00,* May 2000, pp. 93–99.

[39] 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, *Proc. Natl. Acad. Sci. USA* **93** (1996), 5745–5752.

[40] Y. Tanaka, M. Sato, M. Hirano, H. Nakada and S. Sekiguchi, Resource Management for Globus-based Wide-area Cluster Computing, *First IEEE International Workshop on Cluster Computing* (*IWCC'99*), 1999, pp. 237–244.

[41] http://matsu-www.is.titech.ac.jp/cluster-team/.

[42] R.D. Braun, H.J. Siegel, N. Beck, L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen and R.F. Freund, A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Com-

puting Systems, in: *Proceedings of the 8th Heterogeneous Computing Workshop* (*HCW'99*), April 1999, pp. 15–29.
[43] M. Mitzenmacher, How useful is old information, in: *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, 1997, pp. 83–91.