Simgrid: a Toolkit for the Simulation of Application Scheduling

Henri Casanova
Computer Science and Engineering Department
University of California, San Diego
casanova@cs.ucsd.edu

Abstract

Advances in hardware and software technologies have made it possible to deploy parallel applications over increasingly large sets of distributed resources. Consequently, the study of scheduling algorithms for such applications has been an active area of research. Given the nature of most scheduling problems one must resort to simulation to effectively evaluate and compare their efficacy over a wide range of scenarios. It has thus become necessary to simulate those algorithms for increasingly complex distributed, dynamic, heterogeneous environments. In this paper we present Simgrid, a simulation toolkit for the study of scheduling algorithms for distributed application. This paper gives the main concepts and models behind Simgrid, describes its API and highlights current implementation issues. We also give some experimental results and describe work that builds on Simgrid's functionalities.

1. Introduction

Advances in hardware and software technologies have made it possible to use multiple resources concurrently in order to execute multi-component applications. Thanks to improvements in interconnect technologies, it has become practical to distribute applications on increasingly large and dispersed sets of resources: at first within a single system, then among nodes of an MPP system or over multiple clusters, and recently over *Computational Grids* [7]. Consequently, the investigation of scheduling algorithms for distributed applications has been the focus of many research efforts over the years and a large number of such algorithms are now available (see [14, 15] for surveys and comparative studies). This work focuses on algorithms that have the following common objective: assigning a set of *tasks* onto a set of *resources* in a way that is optimal with respect to some metric. A typical goal is to minimize the application *makespan*, that is is the time elapsed between the first application task starts executing and the last task completes. Note that our work is not restricted to that particular metric. The assignment of tasks to resources is called a *schedule*. In this context, a "task" can can be a computation, a data transfer, a data access, etc. and a "resource" can be a CPU, a network link, a disk, etc. Tasks may exhibit dependencies that are commonly described with Directed Acyclic Graphs (DAGs). Scheduling algorithms must then ensure that task precedence requirements are satisfied. The scheduling problem has been shown to be NP-complete [22] in its general form as well as in several restricted cases. Most algorithms use *heuristics* or *guided random searches* to provide approximations of the optimal schedule(s) [14, 15].

The advantage of these methods is that they have low complexity and can thus be used in practical settings. However, it is almost impossible to obtain any analytical result concerning the generated schedules and, in particular, it is difficult to rank scheduling algorithms in terms of schedule quality. The most simple strategy to compare the efficacy of multiple algorithms would be to perform actual experiments by scheduling real applications on real resources. This is not satisfactory for three reasons. First, real applications of interest might run for long periods of time and it is not feasible to perform a statistically significant number of experiments. Second, using real resources makes it difficult to explore a wide variety of resource configurations. Third, variations in resource load over time make it difficult to obtain repeatable results. *Simulation* is then the most viable approach to effectively evaluate scheduling algorithms.

Benchmarking and comparing scheduling algorithms has been recognized as a difficult problem (for instance, see [14] in the context of DAG-structured applications). Most currently available results are obtained with restrictive assumptions. Compute and network resources are often considered homogeneous and assumed to deliver constant performance throughout time. Other assumptions include infinite numbers of available processors, or particular net-

work topologies (e.g. fully connected). Even though some scheduling algorithms [8, 13, 5, 21] specifically focus on heterogeneous environments, it is almost always assumed that a scheduling algorithm is able to obtain perfect prediction concerning task execution times on the resources.

These assumptions are difficult to conciliate with real systems. Network topologies in real system are usually more complex than the ones simulated in most scheduling work. Also, current applications (e.g. applications on the Computational Grid) use resources that have dynamic performance characteristics because they are time- or space-shared. Furthermore, due to this dynamic aspect, it is not possible to obtain exact predictions for task execution times and actual implementations of scheduling algorithms will use inaccurate predictions. The impact of such inaccuracies on the efficacy of the various scheduling algorithms has not been the object of extensive studies. Hence, there is a clear need for new simulation tools that allow for more realistic resource models and for prediction error modeling. A few simulation packages are available [6, 12, 4, 20], but they are not targeted to the simulation of distributed applications for the purpose of evaluating scheduling algorithms. These tools are often very complete and sophisticated but too complex and low-level for our our intended purpose.

In this work we introduce a new package, Simgrid, which provides core functionalities that can be used to build simulators for the study of application scheduling in distributed environments. The goal of Simgrid is to become a useful tool within the scheduling research community. In this paper, we wish to show that Simgrid (i) provides the right model and level of abstraction for its intended purpose; (ii) makes it possible to rapidly prototype and evaluate scheduling algorithms; (iii) allows for more realistic simulation than what is done in previous application scheduling research works; (iv) generates correct and accurate simulation results. Simgrid is a clear improvement over current practice in the application scheduling community. Nevertheless, large-scale computing environments (e.g. Computational Grids) can be extremely complex and the current Simgrid implementation cannot capture all the phenomena involved (e.g. network errors, software overheads, batch resource behavior, complex network topologies, etc.). We will therefore describe ways in which Simgrid can be improved so that it is able to model increasingly complex and realistic computing environments.

This paper is organized as follows. Section 2 describes the Simgrid model and software. Section 3 presents experimental validations and evaluations. Section 4 describes how Simgrid can be used as well as current Simgrid applications and Section 5 contrasts Simgrid with two related projects: Microgrid [18] and Bricks [19]. Section 6 concludes the paper with a summary and a description of future work.

2. Simgrid

2.1. Basic Concepts

Simgrid provides a set of core abstractions and functionalities that can be used to easily build simulators for specific application domains and/or computing environment topologies. Simgrid performs *event-driven* simulation. The most important component of the simulation process is the resource modeling. The current implementation assumes that resources have two performance characteristics: *latency* (time in seconds to access the resource) and *service rate* (number of work units performed per time unit). Simgrid provides mechanisms to model performance characteristics either as constants or from **traces**. This means that the latency and service rate of each resource can be modelled by a vector of time-stamped values, or trace. Traces allow the simulation of arbitrary performance fluctuations such as the ones observable for real resources. Fortunately, traces from real resources (e.g. CPUs, network links) are available via various monitoring tools [24, 9, 11]. In essence, traces are used to account for potential *background load* on resources that are **time-shared** with other applications/users. This trace-based model is not sufficient to simulate all behaviors observable in real distributed systems. However, this first implementation is a first step in that direction and is already a great improvement over current simulation practice for scheduling algorithms evaluation. Section 6 discusses possible ways to enhance Simgrid's resource model so it becomes more realistic.

An important question is that of the **topology** of resource interconnections. A fully-connected topology is often assumed in scheduling algorithm works. Non-fully-connected topologies can then be seen as special cases of the fully-connected topology, and are addressed by some scheduling algorithms [17, 16, 23]. One way to implement Simgrid would have been to enforce a fully-connected topology (e.g. one network link for each pair of computing resources). The user could then specify which connections are actually enabled. This is typically implemented with a connection matrix where some entries are zeroed out. There is one major drawback to that approach: the fully-connected model fails to model realistic distributed computing environments. In a real environment, network links are usually shared by communication tasks. Consider two distinct local area networks, say A and B, on the Internet. Two processors in A communicating with two processors in B typically share a common Internet link. The fully-connected model assumes dedicated links between communicating processors and would over overestimates the achieved data transfer rate by a factor of 2 in our example. Along the same lines, the fully-connected model prevents the simulation of complex network topologies with multiple network links between

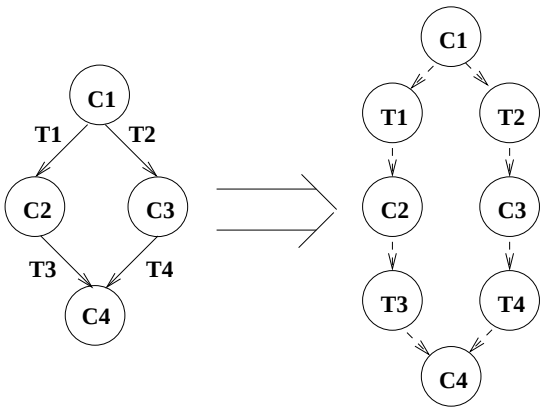

Figure 1. Simgrid Task Model

any two resources (e.g. with intermediate routers). In the Simgrid approach, no interconnection topology is imposed. Instead, Simgrid treats CPUs and network links as unrelated *resources* and it is the responsibility of the user to ensure that his/her topology requirements are met. For instance, one can create multiple links in between hosts (or group of hosts) to simulate the behavior of simple routers. This approach is very flexible and makes it possible to use Simgrid for simulating a wide range of computing environments.

In addition, Simgrid does not make any distinction between data transfers and computations: both are seen as **tasks** and it is the responsibility of the user to ensure that computations are scheduled on processors and file transfers on network links. Figure 1 shows an example of an application with four computational tasks (C1, C2, C3, C4) and four data transfers (T1, T2, T3, T4). On the left side is the traditional DAG representation where edges symbolize both data transfers and task dependencies. In the Simgrid model, on the right side, edges symbolize task precedences and additional nodes are inserted for data transfers. Even though there is little conceptual difference between the two models, we will see in Section 2.2 and in the appendix that having a more abstract notion of task greatly simplifies Simgrid's implementation and makes its API more concise. Note that in its current version Simgrid assumes that computational tasks are CPU-bound and that data transfer tasks are bandwidth-bound.

As mentioned in the introduction, scheduling algorithms often use task execution time **prediction** to perform the assignment of tasks to resources. Simgrid provides a way to simulate prediction with arbitrary error behavior. It thus makes it possible to evaluate scheduling algorithms not only under the traditional assumption of perfectly accurate prediction, but also under a variety of prediction error scenarios. In fact, it is possible to simulate realistic prediction errors such as the ones experienced when using deployed monitoring/forecasting systems like the NWS [24].

2.2. API

Simgrid provides a C API that allows the user to manipulate two data structures: one for resources (*SG_Resource*) and one for tasks (*SG_Task*). A resource is described by a name, a set of performance related metrics and traces or constant values. For instance, a processor is described by a measure of its speed (relative to a reference processor), and a trace of its availability, i.e. the percentage of the CPU that would be allocated to a new process; a network link is described by a trace of its latency, and a trace of its available bandwidth. A task is described by a name, a *cost*, and a state. In the case of a data transfer the cost is the data size in bytes, for a computation it is the required processing time (in seconds) on the reference processor. Finally, the state is used to describe the task's life cycle (*not scheduled*, *scheduled*, *ready*, *running*, *completed*). Simgrid's API provides basic functions to operate on resources and tasks (creation, destruction, inspection, etc.), functions to describe possible task dependencies, and functions to schedule and un-schedule tasks to resources. Simgrid also provides a function to simulate performance prediction: *SG_getPrediction()*. That function can be used to simulate standard prediction error behaviors (perfectly accurate, uniformly distributed errors with given bounds, etc.). In addition, the user can pass to *SG_getPrediction()* a pointer to a function that implements any arbitrary prediction error model.

Only one function is necessary to run the simulation: *SG_simulate()*. This function leads tasks through their lifecycle and simulates resource usage. It is possible to run the simulation for a given number of (virtual) seconds, or until all/any/one task(s) complete(s). In all cases, *SG_simulate()* returns a list of tasks that have completed since the last time it was called. The *SG_getClock()* function returns the simulation's virtual global time and can be called at any time during the simulation. It is possible to perform post-mortem analysis of the application's execution by inspecting task start and completion times, as well as the resources that were used. One can then compute several metrics concerning the simulation. Common examples are resource utilization, task completion rates, or overall application makespan. A simple example program using the Simgrid API is shown in the appendix. The following section discusses some implementation issues.

2.3. Implementation

The implementation of Simgrid (including a test suite) consists of 10,000 lines of C. Optimizations throughout the code aim at improving memory usage and speed. A few simple techniques are used to reduce the memory space used by traces: traces can be shared by resources; sections

of large traces are loaded only when needed and discarded when un-used. When so many resources are to be simulated that the amount of memory needed still exceeds the available RAM, it is then possible to coarsen the grain of the traces (by averaging contiguous time-stamped values) by several factors. However, this results in loss of accuracy (see Section 3 for quantitative results).

Most of the CPU time during a Simgrid simulation is spent processing traces (usually integrating trace values over time intervals). Results presented in the rest of this paper use traces with time-stamps that are at least 10 seconds apart. This is consistent with standard traces obtained via tools such as the NWS [24]. A way to speed up trace processing is to build *multi-level* traces, where each additional level uses a coarser time-scale. Integrations over a given time interval can then be performed primarily on large time-scales, and on progressively finer time-scales close to the edges of the time interval. This simple optimization leads to improvements of several orders of magnitude. Experiments conducted on a Pentium II 360MHz showed that the ratio between Simgrid CPU time and virtual time simulated for a task on a resource is of the order of 10^{-6} when single-level traces are used. This means that three-hundred hours of sequential computation time can be simulated in about 1 second. Using traces with 4 time-scale levels typically lowers the ratio of Simgrid CPU time over simulated time to the order of 10^{-10}. Maintaining multiple-level traces leads to an extra memory cost, but that cost is well worth the improvement in speed as seen in what follows.

Trace processing occurs during calls to *SG_simulate()* and to *SG_getPrediction()*. Simulated predictions of task execution times are typically based on tentative task simulations. For instance, to simulate predictions that exhibit a $\pm 10\%$ uniformly distributed error it is necessary to first simulate a task to obtain an exact prediction, and then to add uniformly distributed random noise to that prediction. Many scheduling algorithms make heavy use of prediction. Let us consider the problem of scheduling n independent tasks on p resources in the presence of task affinities (see [15] for a study of such scheduling problems), and say that one wants to simulate a scheduling algorithm which uses task completion times as priorities for list scheduling. A simple computation shows that the total number of predictions required is $n(p + 0.5n + 0.5)$. Assuming 1,000 tasks, 30 resources, and an average task execution time of 300 seconds, one can then estimate Simgrid's CPU time on a Pentium II 360 MHz to be of the order of 160 seconds using single-level traces, or of 0.016 seconds with multiple-level traces. The use of multiple-level traces clearly makes it feasible to run a large number of experiments with different settings in order to obtains statistically significant results.

3. Experimental Validation

In this section we perform the following experiment: we simulate an application on resources described by *synthetic* traces that allow us to compute the application's makespan analytically. This allows us to validate the Simgrid implementation by showing that the relative errors between analytical and simulated makespans are small.

In these experiments we use a master-worker application where n work units need to be performed. Scheduling is done with a self-scheduling workqueue algorithm [10]. Resources consists of $p + 1$ processors (p workers and one master) and p network links (one link between each worker and the master). All traces are piece-wise linear so that it is straightforward to compute task execution times analytically. We performed 10,000 runs of Simgrid for different values of p and n. Experiment parameters are sampled from uniform distributions as follows: Let $U(x, y)$ denote the uniform probability distribution on the interval $[x, y]$. The costs of computational tasks are $U(600, 3600)$ in seconds, data transfer sizes are $U(0.1, 200)$ in MBytes, and the host relative speeds are $U(0.5, 1.5)$. CPU traces are generated by sampling processor availability from $U(1, 100)$ as a percentage, placing the availability values $U(30, 300)$ seconds apart on a time line, and interpolating them by a piece-wise linear function. The similar procedure was used for network bandwidth with values sampled from $U(0.5, 100)$ in Mbits/sec.

We performed runs for values of p between 2 and 30 and for n/p between 10 and 50. Table 1 shows that the average relative simulation errors on the makespan increases with the total number of simulate resource usage hours. For instance, the values $p = 30$ and $n/p = 50$ denote an application that consists of 1,500 computational tasks and 3,000 data transfers and where the average number of simulated CPU and data transfer hours was on the order of 2,500 in the simulated computational environment. Errors due to the *discretization* of the piece-wise linear traces accumulate and become significant for such long-running simulation applications.

Because of the use of multi-level traces (see Section 2.3) the degree of discretization has virtually no impact on Simgrid's performance. Note that trace coarsening may still be needed due to available memory constraints. The impact of such coarsening on simulation accuracy is difficult to assess as it depends on the "shape" of each individual trace. A simulation of resources with slowly varying performance metrics will tolerate traces that have large time-scales. On the other hand, finer time-scales are needed to accurately capture more erratic behaviors. In our example, with piece-wise linear traces, the makespan relative error grows exponentially with the coarsening factor. Better accuracy than the ones shown in Table 1 can be achieved by using finer

Table 1. Average relative errors for ranges of simulated resource usage

Simulated hours of resource usage	Average relative makespan errors
0-25	0.09 %
25-50	0.13 %
50-100	0.20 %
100-250	0.28 %
250-500	0.46 %
500-750	0.90 %
750-1000	1.38 %
1000-1500	1.80 %
1500-2000	3.41 %
2000-2500	4.20 %

discretizations (e.g. one value every second). However, typical traces generated by currently available monitoring tools contain measurements at least 10 seconds apart.

Note that the main purpose of Simgrid is the comparison of scheduling algorithms. The errors caused by discretization are random and should have similar statistical impacts on different simulations. We believe that taking averages over sufficient numbers of experiments still allows for accurate comparison of scheduling algorithms even though simulated makespans deviate from actual makespans by a few percents due to trace discretization. Further validation of Simgrid's ability to produce accurate simulation results will be given in future papers where Simgrid is used in a particular context (see Section 4.2). Preliminary experiments in [3] provide evidence of the ability of Simgrid to rank algorithms accurately, and future work will report on more extensive experimentations.

4. Building Simulators with Simgrid

4.1. Simulating Scheduling Algorithms

The simulation of scheduling algorithms requires support for setting up the simulated application and computing environment, implementing the algorithm itself, and simulating the application execution on a set of resources. The Simgrid library offers functions that can be used for those purposes. As seen in Section 2.2 and in the appendix, building the simulated application (with task dependencies) and computing environment involves simple calls to API functions. Simulating the application execution once its tasks have been scheduled is also rather straightforward as it involves a single function (*SG_simulate()*). The implementation of the scheduling algorithm itself is the only part of the simulation that must be designed with care.

We distinguish two types of scheduling algorithms: *compile-time* and *run-time*. With compile-time algorithms, all scheduling decisions are made before the application starts. Those decisions are usually based on performance prediction for task running times and a variety of prediction behaviors can be simulated with *SG_getPrediction()*. Each scheduling decision is implemented with a call to *SG_scheduleTaskOnResource()*. Traditionally, scheduling algorithms assign computational tasks to computational resources and it is then the responsibility of the Simgrid user to ensure that possible data transfers are scheduled on appropriate network links. Once all tasks (computations and transfers) have been scheduled, a single call to *SG_simulate()* can run the application until all tasks have completed. Note that *SG_simulate()* detects infinite simulation time and returns with an error code.

Run-time algorithms are a little more challenging as scheduling decisions can be modified while the application is running. The rationale is that the scheduling algorithm can then adapt to changes in the computing environment and refine the schedule. At times, control must then be returned from *SG_simulate()* to the scheduling algorithm itself. For implementing self-scheduling algorithm such as a workqueue, *SG_simulate()* allows control to be returned each time a task completes. For other adaptive algorithms, one can specify how long the simulation should until control is returned. *SG_simulate()* returns with the list of tasks that have completed since the last call and one can call *SG_unScheduleTask()* to modify scheduling decisions concerning tasks that have not started execution or that are currently running. Simgrid provides many simple functions that help the user keeping track of past scheduling decisions so that they can be modified. An interesting question is that of pending data transfers and whether they should be cancelled (at the cost of having wasted bandwidth) or maintained. If they are maintained, then it may be necessary to reschedule computational tasks using the transfered data on the computational resource that is the destination of the transfer. Results experimenting with several strategies to implement run-time scheduling algorithms will be presented in a future paper.

As explained in the following section, we have successfully implemented a large number of compile- and run-time algorithms found in the literature. Even though some work is required to implement those algorithms, we believe that the current Simgrid API allows for the required flexibility for the purpose of scheduling algorithm research.

4.2. Examples of Simgrid Applications

At the time this paper is being written, two simulators have been implemented using Simgrid. This section pro-

vides brief descriptions of these two projects and draws conclusions concerning usability.

The first application of Simgrid is a simulator, PSTSim, whose goal is to evaluate scheduling strategies for *parameter sweep applications* over the Computational Grid. The Grid model is as follows: a set of *sites* where hosts are available for computation; sites are interconnected among each others with network links; hosts within a site have access to a shared storage device and there is no other intra-site communication. The applications are usually long-running and consist of large numbers of independent tasks with the added complexity that tasks may share large input files. The scheduling problem is then to place those files strategically in shared storage so as to minimize data transfers. A full description of the models and of the scheduling issues can be found in [1]. In that paper, PSTSim was used to compare 5 scheduling algorithms over a large space of Grid configurations and application specification. The results obtained led to an implementation of some of those scheduling algorithms in a user-level Grid middleware package, APST, described and evaluated in [3].

The second simulator based on Simgrid, DAGSim, is focused on evaluating scheduling algorithms for applications structured as DAGs. The model is quite different from the previous simulator, both in terms of computing environment and application, but Simgrid's flexible approach was equally amenable to both simulators. We focused on a number of well-known algorithms (DLS, DSC, ETF, MCP, HDLFET as listed in [14]) and implemented adaptive versions for each one of them. Our current interest is in studying the impact of resource performance prediction errors on those algorithms in order to determine which ones are are practical in Computational Grid settings. Simgrid provides the appropriate model to simulate various prediction error behaviors (see Section 2.2) and a future paper will present results obtained with DAGSim.

Both PSTSim and DAGSim were straightforward to implement because Simgrid provides all core functionalities that address the simulation details. Therefore, we were able to focus mostly on the implementation of the scheduling algorithms. Based on this experience we claim that Simgrid provides the right level of abstraction for rapidly building simulators that target scheduling algorithm evaluation.

5. Related Work

Among the research projects that focus on simulating distributed applications, Microgrid and Bricks are the most related to our work. Microgrid [18], developed at UCSD, aims at providing a *virtual grid* infrastructure for the study of Grid resource management issues. Microgrid is more an emulator than a simulator in that it allows applications to run on top of the virtual grid as is. At the moment, Microgrid targets the Globus environment. A virtualization layer intercepts calls from the application to emulate them on real resources while mimicking the performance characteristics of virtual resources. For instance, this makes it possible to study the behavior of a given Globus application in any Grid configurations while running the emulation on a homogeneous cluster. In that sense, Microgrid enables repeatable, controllable experimentation with dynamic resource management techniques. The concept of a virtualization layer allows for very accurate simulations.

One major difference with our work is that Microgrid is an emulator, meaning that actual application code is executed on the virtual Grid. We have seen in Section 1 that the evaluation of scheduling algorithms requires a large number of experiments in order to obtain statistically significant results. Running such large numbers of experiments with Microgrid is not possible since Microgrid simulations will in most cases require more time to complete than the application would on the simulated Grid. Another fundamental difference is that Microgrid targets existing Globus applications whereas our work targets scheduling algorithm designers who work only with an application model. Using Microgrid would require a Globus implementation of the application model targeted to the virtual Grid. Clearly, Simgrid and Microgrid take rather different approaches and focus on different issues. Note that the results obtained with Microgrid are undoubtedly more precise than the one obtained with Simgrid as all aspects of the implementation are modelled (e.g. Globus software overhead, authentication cost, etc.). In addition, Microgrid uses a sophisticated network simulator. We are currently investigating the re-use of that simulator within Simgrid in order to achieve more realistic network modeling.

The Bricks [19] project is developed at the Tokyo Institute of Technology and focuses on simulating scheduling policies for *Global Computing Systems* over the Computational Grid. In the Bricks context, Global Computing Systems [2] are client-server systems that provide remote access to scientific packages or libraries. Essentially, Bricks is a simulator for resource allocation strategies and policies for multiple clients, multiple servers scenarios. Our work departs from Bricks primarily because we target scheduling algorithms for a single structured application, as opposed to scheduling policies for multi-user systems where all requests for computations are independent: we are interested in application makespans as opposed to average overall service rates. An interesting path to explore would be the possible re-use of some of Bricks' components (e.g. CPU and network simulation components) within Simgrid.

6. Conclusion and Future Work

In this paper we have introduced Simgrid, a simulation package that provides basic functionalities for the study of scheduling algorithms for parallel applications in distributed environments. We showed that, in addition to providing the appropriate level of abstraction, Simgrid produces accurate simulation results and is efficient. Even though Simgrid has already proven to be a valuable tool (see Section 4), several limitations still need to be addressed.

The current resource model in Simgrid (based on latencies and service rates) is not sufficient to model all aspects of complex environments such as Computational Grids. For instance, a more sophisticated model is needed for space-shared multi-processors accessible via batch-systems, for resource failure, for different kinds of overheads (e.g. encryption, authentication, on-line instruments), for resource contention (the current model assumes fair sharing among all tasks), and for resource failure. In particular, realistic network modeling is challenging and we are currently investigating how available network simulators can be used within Simgrid. The current task model could also be made more general. For instance, the assumption that all computational tasks are CPU-bound is not quite realistic. In fact, it should be possible to describe a task with multi-dimensional cost (e.g. 70% CPU and 30%I/O) and to add performance metrics to simulated CPUs to account for I/O operations. An alternate solution would be to provide a new type of resource (e.g. disk, or NFS-mounted disk) that would allow users to simulate the CPU-bound and I/O-bound part of their application tasks as two separate Simgrid tasks. These possibilities are currently under investigations. All these extension to the current implementation should have little impact on the current API and usage.

Simgrid's current support for application post-mortem analysis is limited to very simple queries. At the moment, it is the responsibility of the user to collect and process all post-mortem data. Better support for obtaining and analyzing information about a run will be provided in future versions. Finally, there is no built-in support for task duplication. Entire classes of scheduling algorithms use task duplication and at the moment, the Simgrid user must emulate duplication by keeping track of and cancelling duplicates when necessary. Developing a simple task duplication API should be rather straightforward and will be included in future versions.

The goal of the Simgrid project is to become a useful tool within the scheduling research community, specifically targeted to the evaluation of scheduling algorithms in Computational Grid settings. Given that goal, the three main objectives must then be *correctness*, *accuracy* and *usability*. We believe that the current Simgrid implementation described and evaluated in this paper has already taken major steps in those directions. Simgrid v1.0 has been released and the webpage at *http://gcl.ucsd.edu/simgrid* contains all relevant information as well as example programs.

Acknowledgement

The author wishes to thank the reviewers as their comments have helped to improve the quality of this paper.

References

[1] 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)*, pages 349–363, May 2000.

[2] H. Casanova, S. Matsuoka, and J. Dongarra. Network-Enabled Server Systems: Deploying Scientific Simulations on the Grid . In *Proceedings of HPC'01*, 2001. submitted for publication.

[3] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of SuperComputing 2000 (SC'00)*, Nov. 2000. to appear.

[4] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the Ptolemy Project. Technical Report ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, July 1999.

[5] H. El-Rewini and T. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *Journal of PArallel and Distributed Computing*, 9:132–153, 1990.

[6] P. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, 1994.

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

[8] C. Gilbert and A. Edward. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), Feb. 1993.

[9] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A Toolkit for Distributed System Performance Analysis. In *Proceedings of the IEEE Mascots 2000 Conference*, Aug. 2000. to appear.

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

[11] http://www.netperf.org.

[12] http://www.threadtec.com/.

[13] M. Iverson, F. Ozgumer, and G. Follen. Parallelizing Existing Applications in a Distributed Heterogeneous Environment. In *Proceedings of the 4th Heterogeneous Computing Workshop (HCW'95)*, pages 93–100, 1995.

[14] Y. Kwok and I. Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms . *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[15] 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)*, pages 30–44, Apr. 1999.

[16] W. Shu and M. Wu. Runtime Incremental Parallel Scheduling (RIPS) on Distributed Memory Computers. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):637–649, June 1996.

[17] G. Sih and E. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, Feb. 1993.

[18] H. Song, J. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: a Scientific Tool for Modeling Computational Grids. In *Proceedings of SuperComputing 2000 (SC'00)*, Nov. 2000. to appear.

[19] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a Performance Evaluation System for Global Computing Scheduling Algorithms. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, pages 97–104, Aug 1999.

[20] A. Terzis, K. Nikoloudakis, L. Wang, and L. Zhang. IRL-Sim: A general purpose packet level network simulator. In *Proceedings of the 33rd Annual Simulation Sumposium*, Apr. 2000. to appear.

[21] H. Topcuoglu, S. Hariri, and M. Wu. Task Scheduling Heuristics for Heterogeneous Processors. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW'99)*, pages 3–14, 1999.

[22] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:434–439, 1975.

[23] W. Willebeek-LeMair and A. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, Sept. 1995.

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

Appendix: Simgrid example program

This simple example simulates an application similar to the one shown in Figure 1 on two processors interconnected by a single network link. This example uses an arbitrary schedule but is sufficient to demonstrate how Simgrid can be used to simulate more complex scenarios.

```
#include "simgrid.h"

int main() {
SG_Resource p1,p2,link;
SG_Task c1,c2,c3,c4; /* Computations */
SG_Task t1,t2,t3,t4; /* Data Transfers */
SG_Task *tasks_done; /* List of completed
tasks */
double clock;

SG_init(); /* Simgrid initialization */

/* Two processors and a network link */
/* P2 twice as fast as P1 */
p1 = SG_createHost("p1",1.0,"./cpu1");
p2 = SG_createHost("p2",2.0,"./cpu2");
link = SG_createLink("link",
"./latency","./bandwidth");

/* Create the tasks */
c1 = SG_newTask("c1",50.00); /* 50 s */
c2 = SG_newTask("c2",100.00); /* 100 s */
c3 = SG_newTask("c3",200.00); /* 200 s */
c4 = SG_newTask("c4",80.00); /* 80 s */
t1 = SG_newTask("t1",1000.00); /* 1 Kb */
t2 = SG_newTask("t2",1000.00); /* 10 Kb */
t3 = SG_newTask("t3",200000.00);/* 200 Kb */
t4 = SG_newTask("t4",200000.00);/* 200 Kb */

/* Set the dependencies */
SG_addDependent(t1,c1);SG_addDependent(t2,c1);
SG_addDependent(c2,t1);SG_addDependent(c3,t2);
SG_addDependent(t3,c2);SG_addDependent(t4,c3);
SG_addDependent(c4,t3);SG_addDependent(c4,t4);

/* Schedule computations to hosts */
SG_scheduleTaskOnResource(c1,p1);
SG_scheduleTaskOnResource(c2,p1);
SG_scheduleTaskOnResource(c3,p2);
SG_scheduleTaskOnResource(c4,p2);

/* Transfers t1,t4 within a single host */
SG_scheduleTaskOnResource(t1,SG_LOCAL_LINK);
SG_scheduleTaskOnResource(t4,SG_LOCAL_LINK);
/* Trasfers t2,t3 scheduled on the link */
SG_scheduleTaskOnResource(t2,link);
SG_scheduleTaskOnResource(t3,link);

/* run the simulation until c4 completes*/
SG_simulate(0.0,c4,SG_VERBOSE);
clock=SG_getClock();

/* run for 100 seconds of virtual time */
SG_reset();
tasks_done=SG_simulate(100.0,NULL,SG_VERBOSE);

/* run until at least one task completes */
SG_reset();
tasks_done=SG_simulate(-1.0,NULL,SG_VERBOSE);
clock=SG_getClock();

/* Free memory used by Simgrid */
SG_shutdown(); exit(0);
}
```