DAGSim: A Simulator for DAG Scheduling Algorithms

Aubin Jarry, Henri Casanova, Francine Berman

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

DAGSim: A Simulator for DAG Scheduling Algorithms

Aubin Jarry
(École Normale Supérieure de Lyon, France)
Henri Casanova and Francine Berman
(Dept. of Computer Science and Engineering,
University Of California, San Diego, U.S.A.)

Dec 2000

Research Report N° 2000-46

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

DAGSim: A Simulator for DAG Scheduling Algorithms

Aubin Jarry
(École Normale Supérieure de Lyon, France)
Henri Casanova and Francine Berman
(Dept. of Computer Science and Engineering, University Of California, San Diego, U.S.A.)

Dec 2000

Abstract

Scheduling the tasks of a distributed application has been an active field of research for several decades. The classic scheduling problem is to find a assignment of application tasks onto a set of distributed resources in a way that minimizes the overall application execution time. This problem can be formulated for different classes of applications and computing environments. It has been shown that most non-trivial instances of the scheduling problem are NP-complete. As a result, many research works propose different solutions including heuristic-based algorithms, guided random searches, and genetic algorithms. However, it is difficult to determine which solutions are practical for real scenarios. Indeed, most available results ignore many characteristic of large computing environments such as dynamically changing resource performance and availability, complex network topologies and network contention, data storage issues, or inaccuracy of resource performance estimates.
In this paper, we present DAGSim, a simulator specifically designed to evaluate scheduling algorithms for application structured as Directed Acyclic dependency Graphs (DAGs). This simulator is built on top of Simgrid [3], a toolkit for the fast and accurate simulation of distributed applications with realistic assumptions concerning the computing environment. We describe the implementation of DAGSim and how it was used to implement two well-known scheduling algorithms (DCP [6] and DLS [7]). After giving preliminary results with these two algorithms, we give implementation recommendations for further improvement of Simgrid and DAGSim.

Keywords: heterogeneous processors,load balancing,grid computing, NWS, scheduling, simulator

Résumé

Beaucoup de travaux de recherche proposent des solutions différentes au problème d'ordonnancement dont diverses heuristiques, des investigations aléatoires, et des algorithmes génétiques. Cependant, il est difficile de déterminer les solutions utilisables dans des scénarios réels. En effet, la plupart des résultats disponibles font abstraction de beaucoup des caractéristiques d'une grande grille de calcul, notament l'évolution dynamique des performances et de la disponibilité des ressources, la complexité topologique du réseau et la gestion des flux, le problème du stockage des données, ou encore les erreurs sur l'estimation des performancves des ressources.
Dans ce rapport nous présentons DAGSim, un simulateur spécifiquement créé pour évaluer les algorithmes d'ordonnancement pour des applications structurées en Graphes oriéntés acycliques de dépendances(DAG). Ce simulateur est construit au-dessus de Simgrid [3], une bibliothèque réalisant des simulations rapides et pertinentes d'applications distribuées sur un environnement réaliste. Nous décrivons comment DAGSim a été implémenté, son utilisation pour implémenter deux algorithmes d'ordonnancement connus (DCP [6] et DLS [7]). Nous donnons quelques résultats sur ces deux algorithmes avant de suggérer des recommandations pour le développement futur de Simgrid et de DAGSim.

Mots-clés: processeurs de vitesses différentes, équilibrage de charge, grid computing, NWS, ordonnancement, simulateur

1 Introduction

Fast networks have made it possible to aggregate distributed CPU, memory, and storage resources into Computational Grids [4]. The expectation is that such systems will provide the potential for application performance superior to that achievable on any single system. However, achieving such performance on the Grid is a non trivial issue for parallel applications whose performance is highly dependent upon the efficient coordination of their constituent components. An active field of research has then be to provide software that schedules the user's application on his/her behalf [2, 1]. The important issue is then the choice of the scheduling algorithms that are practical in real computing environments. In this paper we target applications that are structured as *Directed Acyclic Graphs* (DAGs).

A wealth of compile-time algorithms exist for such applications. However, it is difficult to decide which ones should be employed on the Grid. Indeed, most available evaluation results use assumptions that have become increasingly unrealistic for modern computing environments. Studying the behavior of scheduling algorithms on real platforms, though providing accurate and undoubtedly realistic results, is not practical due to the dynamic availability of resources. In particular, experiments are often non-reproducible, which prohibits fair comparison among different scheduling algorithms. It is then necessary to resort to simulation. As a result, the simplest scheduling algorithms are often used for Grid applications (e.g. self-scheduling [5]), while the more sophisticated ones are usually left aside, existing in theory only, or targeted to homogeneous dedicated systems (multi-processor platforms). This is often due to the fact that sophisticated scheduling algorithms require performance prediction. It is then clear that a new set of simulation results must be obtained for realistic scenarios, e.g. with dynamically changing resource performance and availability, complex network topologies and network contention, data storage issues, and/or inaccuracy of resource performance estimates.

Our goal is to provide a simple and adequate simulation framework for evaluating scheduling algorithms in realistic Grid environments. This paper is structured as follows. Section 2 describes the implementation of DAGSim. Section 3 introduces two scheduling algorithms that have been used for validating DAGSim and Section 4 present initial results. Finally, Section 5 concludes this paper with a discussion of future improvements and directions.

2 DAGsim

In this section we describe the implementation of DAGSim version 1.0. First we introduce Simgrid, DAGSim's underlying simulation toolkit.

2.1 Simgrid

Simgrid [3] is a simulation toolkit that provides core functionalities that can be used to build simulators for distributed applications in distributed environments. It allows for building simulators for specific application domains and/or computing environment topologies. Simgrid performs **trace-based** simulation. This means that the performance metrics of each resource are modelled by vectors of time-stamped values, or *traces*. Traces allow the simulation of arbitrary dynamic performance fluctuations such as the ones observable for real resources. Traces are used to account for potential *background load* on resources that are **time-shared** with other applications/users.

An important question is that of the **topology** of resource interconnections. 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 calling software to ensure that its topology requirements are met. This approach is very flexible and makes it possible to use Simgrid for simulating a wide range of computing environments.

Sophisticated 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 [8].

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. 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. The possible states are *not scheduled*, *scheduled*, *ready*, *running*, and *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 life-cycle and simulates resource usage. It is possible to run the simulation for a given number of (virtual) seconds, or until at least one task completes. In both cases, `SG_simulate()` returns a list of tasks that have completed. The `SG_getClock()` function returns the simulation's virtual global time. 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. Further information on Simgrid is available at http://gcl.ucsd.edu/simgrid.

2.2 Implementation of DAGSim v1.0

Definition 1 (DAG application) *A DAG (Directed Acyclic Graph) application is a set of tasks which has precedence constraints, modelled by a direct acyclic graph: each task is represented by a node in the graph, the dependencies by the edges. A cost is assigned to each node (computational cost of the task) and to each edge (the amount of data to transfer to the next task).*

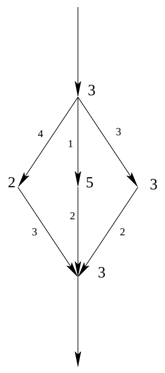

Figure 1: DAG application

The DAG Simulator (or DAGSim for short) is implemented on top of Simgrid, and is dedicated to general DAG applications, and DAG scheduling algorithms. The goal of DAGSim is to provide a simple interface for implementing DAG scheduling algorithms and for evaluating these algorithms in various simulation scenarios (thanks to the functionalities of Simgrid).

In its current incarnation, DAGSim takes two files as input: a file describing the application (DAG file), and a file describing the computing environment to simulate (Grid file). DAG files can be taken from real applications or automatically generated (e.g. with standard graph generation techniques) and Grid files are usually inspired by real topologies. Once the DAG and Grid files have been parsed, DAGSim uses the following internal data structures:

- DAG application: A dag application is represented as an array of nodes where each node is described by the following fields:
  - global index: this is the position in the array of nodes. This position may be used as an identifier when using other arrays representing special attributes for the node, depending on what attribute are relevant for the scheduler
  - node type: the DAG contains two special nodes, ROOT from which every path begins, and END where every path ends. Those nodes are useful for recursive or straightforward exploration of the graph. Standard nodes are tagged COMPUTATION (they represent a computational task). Data dependencies are tagged TRANSFER (with a communication cost), and have only one parent (the from node) and one child (the towards node).
  - parents: the number of parents and an array of these nodes are provided (the ROOT node is the only one without any parents, TRANSFER nodes have only one parent).
  - children: the number of children and an array of these nodes are provided (the END node is the only node without any child, TRANSFER nodes have only one child).
  - cost: the computational cost for a COMPUTATION node, or the message length for a TRANSFER node. ROOT and END nodes may not have any cost; if they have, they are assumed to be computational. These costs are consistent with the Simgrid model.
  - abstract task: a pointer to the Simgrid abstract task represented by the node, used when calling predictions or actual scheduling

- Grid: The Grid is represented as an array of hosts, an array of links, and a communication matrix binding links and hosts
  - host : each processor has a static speed and a load evolving dynamically. The load is not to be immediately known by the user, but is passed to Simgrid, which in return will give predictions to the user.
    - global index: position in the array of hosts; this is also used as an identifier for the communication matrix binding links to hosts.
    - relative speed: relative speed of the processor: 1.0 should be an average speed for the set of all processors.
    - resource: the corresponding Simgrid Resource, used for calls to Simgrid
  - link : each link has a static latency, a static bandwidth, and a dynamic load, all of which are hidden. The user is required to use Simgrid for predictions calls
    - global index: position in the array; this is also used as an identifier in the communication matrix.
    - resource: the corresponding Simgrid Resource, used for calls to Simgrid
  - communication matrix: This matrix represent the topology of the grid: entries are the indexes of two hosts, and elements point either to a link or to `void` (when no communication link is available between two hosts). While this matrix is usually symmetric, it is not necessary, $M[i, j]$ being the link used while sending a message from host indexed i to host indexed j. If the diagonal is not specified by the user (in the Grid file) it is filled by a *local link*, Simgrid's abstraction for infinitely fast link (local communications are done instantly). Also note that a same link can be used several time in this matrix.

2.2.1 Implementing a Scheduling Algorithms within DAGSim

One of the goals of DAGSim is to make it easy to implement and evaluate a large number of scheduling algorithms. When implementing a scheduling algorithm, the user is required to comply with the following format, which will be used by DAGSim. The scheduling algorithms are expected to be able to be be called several times while the tasks are completing. This allows for *dynamic scheduling* and allows scheduling algorithms to use latest (and most accurate) performance predictions.

- `name`: the name by which the scheduler will be called,
- `comment`: a comment used in the help outputs,
- `init` / `clear`: to functions destined to allocate and to free memory before and after the scheduler is being used,
- `run`: the actual scheduling, of application tasks on resources,
- `re_init`: used to unschedule (or not) tasks not completed, and prepare the tasks in the middle of execution for a new run of the scheduling algorithm.

Each new scheduling algorithm will require data items to reflect the properties of the network and of the task graph it will exploit. These new data items, or attributes, will be linked to existing data structures such as DAGSim nodes, links and hosts. Keep in mind that another scheduler may have been used before this scheduler gets the job.

One important point for most scheduling algorithm is performance prediction. Fortunately, simgrid provides the adequate functionalities. However, note that when making predictions, SIMGRID keeps track of them to avoid predicting two different values for the same input (e.g. when random prediction errors are simulated). However, Simgrid does not check all dependencies for predictions, which heavily depend on the task graph, network connectivity, and the the algorithm itself. Therefore, when simulating large prediction errors, there may be some inconsistencies in repeated predictions (such as a task finishing earlier than one of its children). It is the responsibility of the scheduling algorithm to resolve such inconsistencies.

Apart from re-initializing the specific data of the scheduler, the `re_init` function is responsible for unscheduling tasks that have not completed, and to prepare data for a new run of the scheduling algorithm. The un-scheduling is not built-in in DAGSim, for the scheduler may want to do it in a proper order while updating its specific data. Two points require care when implementing `re_init`: what to do with running tasks (if they are not unscheduled which seems reasonable, the algorithm has to deal with their running state), and what to do with communication tasks that have completed or that are still running. It is the responsibility of the implementor of each scheduling algorithm to make appropriate choices.

Finally, when using a scheduler, it is crucial to give a proper network and a proper DAG. This may seem obvious, but many schedulers are designed for a specific set of networks, and/or a specific set of applications. Testing those schedulers with inappropriate inputs may be interesting for testing purposes, but is likely to cause malfunctions, since the assumptions of the algorithm are no longer correct. Frequent assumptions are : the network is fully connected; there is always a data transfer between two computations, etc...

3 Two scheduling algorithms

In this work, we experimented with two different scheduling algorithms: Dynamic Critical Path scheduling, and Dynamic Level Scheduling. We introduce both algorithms in the following sections.

3.1 Dynamic Critical Path Scheduling

The *Dynamic Critical Path* [6] scheduling algorithm was designed to schedule DAG applications on a dedicated, homogeneous, fully-interconnected network with an infinite amount of processors. Therefore, this algorithm had to be adapted in order to test the relevancy of its principle on a *Grid* (i.e. with heterogeneous, non-dedicated and finite networks). First we briefly explain the original algorithm, thoroughly described in [6], and then point out the modifications that we had to make in the context of DAGSim.

3.1.1 Standard DCP

Definition 2 (Critical Path) *Let G be a direct acyclic graph, $(n_0...n_k)$ a given sequence of nodes. $(n_0...n_k)$ is called a critical path if n_0 is a root node, n_k is an end node, $\forall i \in [0; k-1]$ n_{i+1} is a child of n_i and $\Sigma_{i=0}^{k}$ $cost(n_i)$ + $\Sigma_{i=0}^{k-1}$ $cost(n_i n_{i+1})$ is a maximum. Note: the communication costs are nullified if the two nodes are scheduled and on the same processor.*

Definition 3 (Absolute Earliest Start Time) *The Absolute Earliest Start Time of the Root node is 0. The Absolute Earliest Start Time of any other node n is $max_{parents}(AEST(p) + cost(p) + cost(pn))$. Note: the communication costs are nullified if the two nodes are scheduled on the same processor.*

Definition 4 (Absolute Last Start Time) *The Absolute Last Start Time of the End node equals its Absolute Earliest Start Time. The Absolute Last Start Time of any other node n is $min_{children}(ALST(c) - cost(cn)) - cost(n)$.*

The DCP algorithm proceeds as follows: while some nodes are not scheduled, compute AEST and ALST for all nodes, and schedule the unschedule node with the smallest difference between AEST and ALST (AEST = ALST means that the node is one a critical path). This scheduling minimizes the sum of the node AEST and of the biggest AEST of its unscheduled children, if scheduled on the same processor.

3.2 Modifications for Heterogeneous Environments

Finite Number of hosts The number of hosts in a *Grid* being finite, it may occur that the number of hosts required by the *DCP* algorithm is greater than the actual number of hosts. However, any modification in *DCP* addressing this issue would alter the principle of the algorithm itself, and thus has been discarded. As a result, the *DCP* algorithm will only work with a large-enough grid, or a small-enough application.

Heterogeneity of hosts The heterogeneity of hosts, and moreover their fluctuating performances, will challenge the meanings of AEST and ALST. The proper way to address this problem is to consider the completion time of a task rather than its cost. The immediate problem triggered by this approach is how to cope with those times when the tasks are not yet scheduled : we introduce the concept of virtual Resource.

Virtual resource While not scheduled, in order to perform the necessary computations, each node is considered scheduled to a virtual host with a given speed, and all communication to and from this host are performed by a virtual link with given speed and latency. The proper values to give to those speeds and latency will be discussed in Section 4, while a reasonable first approach uses an average of the available resources.

AEFT Since an earliest start time is not any more equivalent to an earliest completion time in this environment, the AEST has been replaced by the AEFT (Finish Time) in the optimisation part of the algorithm.

Predictions We use predictions to compute the expected completion time of each task. Simgrid allows for simulating inaccurate predictions and this can result in inconsistencies when running the DCP algorithm. In DAGSim, we ensure that inconsistencies are prevented by carefully handling randomness of prediction errors : the computation cost of a given task is predicted only once by sequence of AEST computing, which forbid any inversion in task dependencies (a task can not be started or scheduled before its predecessor has completed).

3.3 Dynamic Level Scheduling

Since the DLS algorithm [7] was originally designed for heterogeneous platforms, the only difference here is the dynamic constraints of a non-dedicated environment.

Definition 5 (Static Level) *The Static Level of the End node is 0. The Static Level of any other node* n *is* $max_{children}(SL(c)) + cost(n)$.

At each step, one computes the set of ready tasks, and the AEFT of each pair (ready task, host). The winning pair is the pair with the maximum (SL(node) - AEFT(node,host)); the winning node is scheduled on the winning host.

Parameters This algorithm uses one implicit parameter, linked with the cost of a node. Such a cost has to be related to the performance of hosts. This is done by choosing an appropriate constant reflecting what "average speed" we expect from the processors. Though not as crucial as the DCP parameters, this changes the algorithm behaviour and is discussed in Section 4.

Predictions As for the DCP algorithm described above, we exert special care when facing inaccurate performance predictions (in simulation).

4 Initial Results

4.1 Parameters of the Scheduling Algorithms

The tuning of the scheduling algorithms' parameters has been made with three goals in mind: avoiding schedule inconsistencies (for DCP), respecting the basic principle for each algorithm, and studying their general behaviour (optimization).

4.1.1 DCP

In DCP, the parameters are the speed of the virtual processor, the speed of the virtual link, and the latency of this link.

Consistency The execution of this algorithm allegedly decrease the overall completion time at each step [6]. This fact is crucial for the correctness of the algorithm, mostly while choosing an adequate processor; indeed, once a task on the critical path is chosen for scheduling, it already has an expected finish time, and its scheduling is supposed to lower its completion time. If not (high virtual speed) the assertions on DCP no longer hold, and the behaviour of the algorithm is not guaranteed. Therefore, a cautious approach consists in setting the respective virtual speeds to the lowest speed in the Grid, without forgetting the dynamic load on the resources. This insures full consistency of the algorithm, but generally lacks in efficiency : the lowest speed can be too low compared to average speed.

Effects of a low virtual speed Using the cautious approach, low virtual speed occurs when the load of a given processor (or link) will cause it to almost stall (mostly for a short period), while the resource is globally more efficient. The low virtual speed will mean that the scheduled tasks will have a short expected completion time compared to the unscheduled one; this causes at each step one ready task to be on the critical path; we have a priority-based List Scheduling Algorithm (critical paths are no longer meaningful).

Behaviour with a higher virtual speed The idea beneath DCP requires that the virtual speeds are close enough to the actual speeds of the resources, regardless of occasional load heights. Indeed, we chose an average speed (on every host, using their loads), and this provides better results, though we can not prove the algorithm behaviour anymore.

4.1.2 DLS

In DLS the single parameters of concern are costs of nodes used to compute the Static Level. The choice of those parameters do not impair DLS stability which is a priority-based list scheduling algorithm. A low cost will promote the ready tasks which can be completed first, while a high cost will disregard the computation time of ready tasks (except for choosing the best host). Here again, an average cost (corresponding to an average speed) seemed the most efficient, with a little lower cost when the loads are changing too much.

4.2 DCP versus DLS

Having very different hosts on a small Grid occur rarely, so this will not generally impair too much DCP. However the relative efficiency of DCP compared to DLS remain highly dependent on the fluctuation of the resource availability. When the fluctuation remain in +/-10% the results given by DCP are comparable to those given by DLS (slightly better or worse, depending on the application), however, the results of DCP will worsen as the fluctuations growth. This is related to the choices of DCP parameters, and to the fact that DCP was not designed for non-dedicated environment. The following results are for randomly generated graphs with 20 computational nodes on fully interconnected networks. The number reported are average application completion times on 100 runs.

Homogeneous processors, homogeneous links, load never greater than 20%

DCP	DLS(average speed)	DLS(0.9×average speed)
375.2	393.1	410.1

Homogeneous processors, homogeneous links, load never greater than 50%

DCP	DLS(average speed)	DLS(0.9×average speed)
587.0	442.6	450.3

Homogeneous processors, homogeneous links, load up to 1.0

DCP	DLS(average speed)	DLS(0.9×average speed)
873.2	491.2	474.4

Heterogeneous processors (speed from 0.7 to 1.4), homogeneous links, load never greater than 20%

DCP	DLS(average speed)	DLS(0.9×average speed)
424.2	387.2	409.8

Heterogeneous processors (speed from 0.7 to 1.4), homogeneous links, load never greater than 50%

DCP	DLS(average speed)	DLS(0.9×average speed)
513.8	451.8	432.1

Heterogeneous processors (speed from 0.7 to 1.4), homogeneous links, load up to 1.0

DCP	DLS(average speed)	DLS(0.9×average speed)
854.0	475.8	475.9

Heterogeneous processors (speed from 0.4 to 1.6), homogeneous links, load never greater than 20%

DCP	DLS(average speed)	DLS(0.9×average speed)
408.5	371.0	366.6

Heterogeneous processors (speed from 0.4 to 1.6), homogeneous links, load never greater than 50%

DCP	DLS(average speed)	DLS(0.9×average speed)
624.6	408.3	388.1

Heterogeneous processors (speed from 0.4 to 1.6), homogeneous links, load up to 1.0

DCP	DLS(average speed)	DLS(0.9×average speed)
883.2	467.7	451.3

These results demonstrate DAGSim's functionalities and are in no way a complete study of the comparative value of DLS versus DCP. Furthermore, for a given Grid, the application graphs plays a great role in the respective performance of the algorithms. So far, we have only used a basic graph generator and a small application set. As discussed in Section 5, our future work will explore a large space of possible DAGs.

5 Conclusion and Future Directions

During its development, DAGSim generated valuable feedback concerning Simgrid's functionalities which led to several improvements within the Simgrid implementation. It appears "random graphs approaches" suffer from the fact that there is no standard graph randomization. Every graph generation algorithm has its own idiosyncrasies, which will affect the type of graphs produced. In turn, the type of graph may drastically affect the performances of the schedulers. For instance, with a highly connected graph, the most effective algorithm will obviously always be the simple sequential one (selecting one processor and schedule everything on it). However, it appears that both DCP and DLS will schedule some nodes on different processors, making some short term improvements (dynamically evolving resources induce changing the order of fast processors), without foreseeing the greatly increased communication tasks created. In the same way one can give an advantage to DCP over DLS in certain condition, by changing some parameters. The main issue being that not only the parameters but the very algorithm itself are influenced by arbitrary choice. Therefore, it is important to use DAGs from applications in various fields of science and engineering. Further research will involve bibliography research on the subject, and development of specific or general graph generators, with better understanding of the underlying distribution laws and their impact on the schedulers' performance.

The main goal of DAGSim is to provide an adequate environment to perform tests with various schedulers on a Grid environment. Now that this environment is available, we are looking forward to implementing other relevant scheduling algorithms and to leading a large set of tests for various applications and Grid configuration.

References

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

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

[3] Henri Casanova. Simgrid: A Toolkit for the Simulation of Grid Application Scheduling. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid'2000)*, 2001. submitted.

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

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

[6] Yu-Kwonk Kwok and Ishfaq Ahmad. Dymamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[7] Gilbert C. Sih and Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, February 1993.

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