A Tool for Prioritizing DAGMan Jobs and Its Evaluation *

Grzegorz Malewicz (malewicz@google.com)

Department of Engineering, Google, Inc., Mountain View, CA 94043, USA

Ian Foster (foster@mcs.anl.gov)

Department of Computer Science, University of Chicago, Chicago, IL 60637, USA and Division of Mathematics and Computer Science, Argonne National Laboratory, Argonne, IL 60439, USA

Arnold L. Rosenberg (rsnbrg@cs.umass.edu)

Department of Computer Science, University of Massachusetts Amherst, Amherst, MA 01003, USA

Michael Wilde (wilde@mcs.anl.gov)

Division of Mathematics and Computer Science, Argonne National Laboratory, Argonne, IL 60439, USA

2000/04/29

Abstract. It is often difficult to perform efficiently a collection of jobs with complex job dependencies due to temporal unpredictability of the grid. One way to mitigate the unpredictability is to schedule job execution in a manner that constantly maximizes the number of jobs that can be sent to workers. A recently developed scheduling theory provides a basis to meet that optimization goal. Intuitively, when the number of such jobs is always large, high parallelism can be maintained, even if the number of workers changes over time in an unpredictable manner. In this paper we present the design, implementation, and evaluation of a practical scheduling tool inspired by the theory. Given a DAGMan input file with interdependent jobs, the tool prioritizes the jobs. The resulting schedule significantly outperforms currently used schedules under a wide range of system parameters, as shown by simulation studies. For example, a scientific data analysis application, AIRSN, was executed at least 13% faster with 95% confidence. An implementation of the tool was integrated with the Condor high-throughput computing system.

Keywords: scheduling, dags, unpredictability, systems, grid, Condor, applications of theory

1. Introduction

Advances in technology have made collections of Internet-connected computers a viable computational platform [10]. Thus, we see many efforts aimed at using multiple distributed computers to solve a single computational problem. Perhaps the major impediment to scheduling complex computations efficiently in this new environment is *temporal unpredictability*:

– Communication takes place over the Internet, and thus may experience unpredictable delays.

– Remote computing workers may not be dedicated to performing the work they receive remotely, and thus may execute that work at an unpredictable rate.

[*] The research of G. Malewicz was supported in part by NSF Grant ITR-800864. The research of I. Foster and M. Wilde was supported in part by the NSF GriPhyN project and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. The research of A.L. Rosenberg was supported in part by NSF Grant CCF-0342417.

This uncertainty in timing virtually precludes accurate identification of critical paths in complex computations, and hence demands a new scheduling paradigm that acknowledges the strengths and weaknesses of the Internet as a computational medium.

A series of recent papers [17, 18, 16] have identified a new goal when scheduling computations that consist of multiple jobs with inter-job dependencies. The goal is to schedule the jobs of a complex computation in a manner that always maximizes the number of jobs that are eligible for assignment to remote workers. In particular algorithms have been developed [16] that can optimally schedule a large family of complex computations. One hopes that this goal has the dual advantage of:

- maximally exploiting available remote workers, by minimizing the likelihood that there is no job for a remote worker; and

- minimizing the likelihood of the "gridlock" that can occur when no new jobs can be assigned pending the completion of already assigned jobs.

The scheduling theory aims to develop a theoretical basis for scheduling complex Internet-based computations, rather than a practical scheduling tool. Indeed, the theory's demands are so great that some computations provably preclude optimal scheduling! The first goal of the current paper is to develop a *practical* scheduling tool that is inspired by the theory but that transcends the theory's limitations—most essentially by being able to handle all computations.

The theory proceeded with an intuitive hope that keeping the number of eligible jobs high increases the utilization of workers and reduces the chances of "gridlock." The second goal of the current paper is to determine to what extent this hope can be realized in practical grid scenarios.

We structure this article as follows. First, we outline the scheduling theory [16] in Section 2, emphasizing the scheduling algorithm. In Section 3.1, we extend that scheduling algorithm to obtain a heuristic that agrees with the theory's algorithm when it works, but that provides a schedule for every computation. In Section 3.2, we describe an implementation of our heuristic as a scheduling tool called PRIO, which we have integrated into the Condor high-throughput computing system [21]. In essence, PRIO prioritizes the interdependent jobs of a given computation encouraging Condor to assign the jobs to workers according to the priorities. Finally, in Section 4, we use simulation studies to evaluate the performance gains achievable from the use of the PRIO tool. We find that for a broad range of situations that one might expect to encounter in Internet-based computing environments, PRIO achieves significant performance improvements.

2. Theoretical foundations

2.1. An Overview

Within the theory [16], a computation is composed of jobs with dependencies, modeled as a *directed acyclic graph* (*dag*, for short) $\mathcal{G}$. Each node of $\mathcal{G}$ represents a component job of the computation. Each arc $(u \rightarrow v)$ of $\mathcal{G}$ represents an inter-job dependency: the execution of job v cannot begin until the execution of job u has completed. We call u a *parent* of v and v a *child* of u. When all parents of an unexecuted job v have been executed, then v becomes *eligible*. The (eligible) jobs of a computation $\mathcal{G}$ are assigned one at a time by a centralized *server*. Each job is assigned to a *worker* who executes the job and returns its results. Upon the return, the job stops being eligible. Time is measured in an event-driven fashion, in which we call the processing of a single job—its assignment, execution, and result-returning—a *step*.

Our goal is to determine a *schedule* Σ for a given $\mathcal{G}$—i.e., an order for assigning jobs to workers—that maximizes the number of eligible jobs *at every step of the computation*. Thus, in a snapshot of the computation wherein t steps have been taken we want the number $E_\Sigma(t)$ of eligible jobs to be maximized, where the maximum is taken over all sets of t executed jobs that honor the job-precedence constraints represented by $\mathcal{G}$'s arcs. Such a schedule is called *Internet-Computing optimal* (*IC optimal*, for short). It is shown [17, 18] that a large class of uniformly structured dags admit IC-optimal schedules. The scheduling algorithm [16] significantly expands this class to a much broader class of dags, which are “assembled” in a uniform way, but whose structures may be far from uniform. However, there do exist even some simple dags whose structures preclude any IC-optimal schedule.

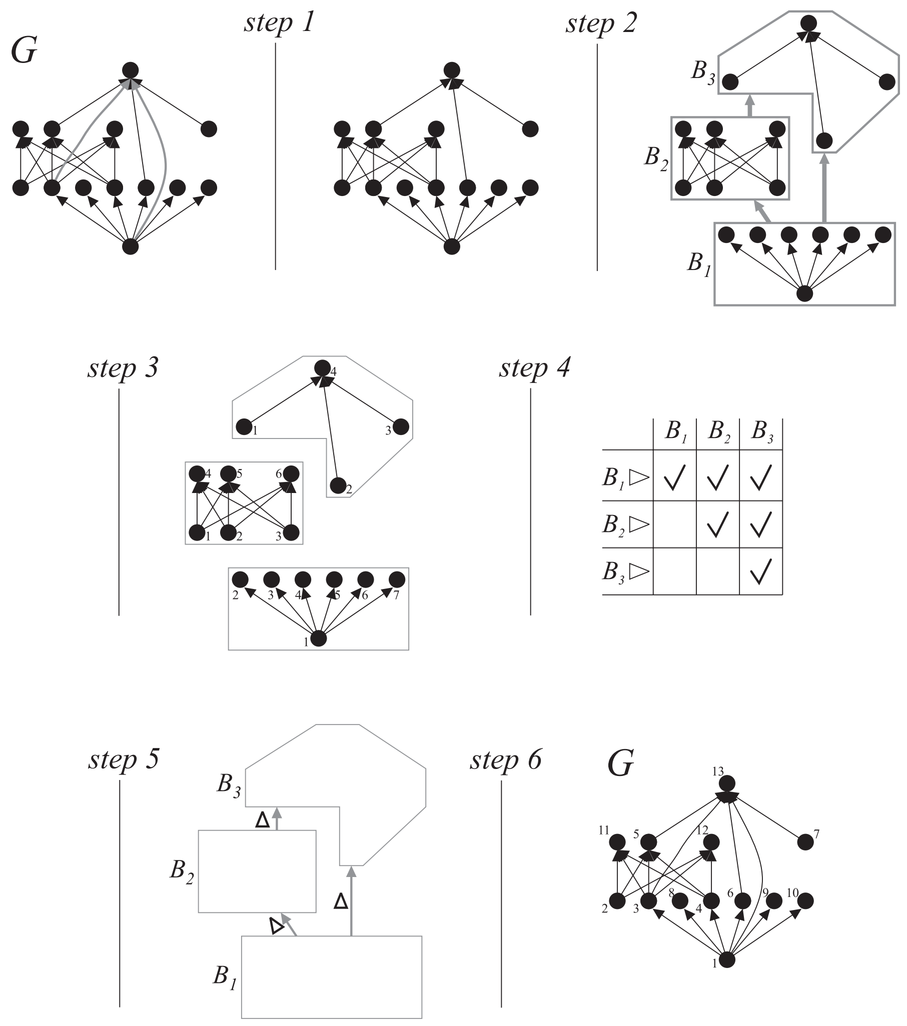
The framework of the preceding three sources idealizes the scheduling problem in a fundamental way. There is the viewpoint that in any snapshot of a computation, at most one remote worker must be served. This idealization allows one to study the scheduling problem formally, with proofs rather than simulations. But the ultimate value of the theory requires that it lead to a practical scheduling regimen that reduces execution times in realistic scenarios. Among other requirements, these scenarios must allow unpredictable numbers of workers to arrive and request jobs at unpredictable times, and it must allow for unpredictable job-execution times.

2.2. The Algorithm

We formulate our scheduling algorithm by adapting the idealized scheduling algorithm [16]. We now outline the structure of the idealized algorithm, which schedules a given dag $\mathcal{G}$ in three phases—divide, recurse, and compose—that match the illustration of Fig. 1. In this article the arcs of the dags depicted are oriented upward. We assume reader’s familiarity with the embedded standard algorithmic notions [8].

The Divide Phase

Step 1. We remove from $\mathcal{G}$ every *shortcut* arc, i.e., every arc $(u \rightarrow v)$ such that v can be reached from u without using the arc. Shortcuts do not affect job eligibility status, but they hamper the



	B_1	B_2	B_3
$B_1 \rhd$	✓	✓	✓
$B_2 \rhd$		✓	✓
$B_3 \rhd$			✓

Figure 1. Illustrating the six steps of the algorithm.

process of decomposing a dag into its "building blocks." Let $\mathcal{G}'$ be the dag obtained by removing all shortcuts from $\mathcal{G}$.

Step 2. We decompose $\mathcal{G}'$ into simpler dags, using a decomposition strategy that reflects our hope that $\mathcal{G}'$ can be composed from *maximal connected bipartite dags* (*building blocks*, for short).

A dag $\mathcal{H}$ is *bipartite* if its node-set can be partitioned into nonempty sets U and V, in such a way that each arc leads from a node in U to a node in V. $\mathcal{H}$ is *connected* if, when arc orientations are ignored, there is a path connecting every pair of nodes in the resulting undirected graph. $\mathcal{H}$ is a *maximal* connected bipartite dag if adding any node of $\mathcal{G}'$ to $\mathcal{H}$ creates a dag that is not bipartite and connected.

We iterate the following process to decompose $\mathcal{G}'$. (1) We identify a building-block subdag $\mathcal{B}$ of $\mathcal{G}'$ all whose sources are also sources of $\mathcal{G}'$. (2) We detach $\mathcal{B}$ by removing from $\mathcal{G}'$ all sources of $\mathcal{B}$ and all sinks of $\mathcal{B}$ that are also sinks in $\mathcal{G}'$. Now, the decomposition process may "fail," because at some point when $\mathcal{G}'$ is not yet empty, the surviving remnant of $\mathcal{G}'$ may not have a bipartite subdag all whose sources are sources of $\mathcal{G}'$. If the process succeeds, though, then the decomposition yields a *superdag* of $\mathcal{G}'$ whose nodes are the building blocks $\{\mathcal{B}_i\}$ that have iteratively been detached from $\mathcal{G}'$, and whose arcs show how the dags need to be composed to yield $\mathcal{G}'$.

The Recurse Phase

Step 3. We attempt to find an IC-optimal schedule Σ_i for each building block $\mathcal{B}_i$. This can often be done because the $\mathcal{B}_i$ often have quite simple structure, certainly much simpler than $\mathcal{G}'$'s. There exist [16, 7] explicit IC-optimal schedules for large families of bipartite dags; Fig. 2 illustrates a representative sample. We simplify the quest for IC-optimal schedules by noting [16] that such schedules can always execute all non-sinks of a dag before any sink. As stated earlier, our quest for IC-optimal schedules may fail for some building blocks.

The Combine Phase

Step 4. We investigate possible "priorities" among the building blocks. Informally, we say that building block $\mathcal{B}_i$ "has priority over" building block $\mathcal{B}_j$ if the schedule that executes all sources of $\mathcal{B}_i$ before any of $\mathcal{B}_j$ renders sinks eligible at least as fast as does any schedule for executing both building blocks. We formalize this notion via the following formal relation. We say that $\mathcal{B}_i \rhd \mathcal{B}_j$ whenever:

- $\mathcal{B}_i$ admits an IC-optimal schedule Σ_i, and $\mathcal{B}_j$ admits an IC-optimal schedule Σ_j;
- the total number of eligible jobs in both dags is always kept maximum if we first execute all sources of $\mathcal{B}_i$ according to schedule Σ_i, then execute all sources of $\mathcal{B}_j$ according to schedule Σ_j, then execute all sinks of $\mathcal{B}_i$ and $\mathcal{B}_j$ in any arbitrary order.

Phrased mathematically, if $\mathcal{B}_i$ has s_i sources, and $\mathcal{B}_j$ has s_j sources, then:

$$\begin{aligned} & (\forall x \in [0, s_i])\ (\forall y \in [0, s_j]) : \\ & E_{\Sigma_i}(x) + E_{\Sigma_j}(y) \ \leq \\ & E_{\Sigma_i}(\min\{s_i, x+y\}) + E_{\Sigma_j}((x+y) - \min\{s_i, x+y\}), \end{aligned} \tag{1}$$

We hope that, for all distinct indices i and j, either $\mathcal{B}_i \rhd \mathcal{B}_j$ or $\mathcal{B}_j \rhd \mathcal{B}_i$. *If* the desired priorities do hold, and *if* the priorities do not conflict with the inter-building-block dependencies induced by the topological placement of the $\mathcal{B}_i$'s in the composite dag $\mathcal{G}'$, *then* by ordering the execution of the $\mathcal{B}_i$'s from one with the highest priority to one with the lowest priority (because $\rhd$ is transitive [16]; $\rhd$ is not reflexive neither antisymmetric) we obtain an IC-optimal schedule for $\mathcal{G}'$—and, thereby, for $\mathcal{G}$.

Step 5. Motivated by the preceding sentence, we check if the superdag respects the inter-building-block priorities. Specifically, we check if, within the superdag, for every node $\mathcal{B}_i$, and every child $\mathcal{B}_j$ of $\mathcal{B}_i$, we have $\mathcal{B}_i \rhd \mathcal{B}_j$. If this holds, then there is hope that the dependencies created by composition do *not* prevent us from executing a job when we would like.

Step 6. By the time we reach this step, we know that $\mathcal{G}$ admits an IC-optimal schedule. We can produce one such schedule as follows. We take any topological sort of the superdag, and sort its components *stably*[1] with respect to the priority relation $\rhd$. We thereby obtain a sequence $\mathcal{B}_{\pi(1)}, \mathcal{B}_{\pi(2)}, \ldots, \mathcal{B}_{\pi(n)}$, of $\mathcal{G}'$'s component building blocks, where π is a permutation on $\{1, \ldots, n\}$. One then obtains an IC-optimal schedule for $\mathcal{G}$ by first executing all jobs that are the sources of $\mathcal{B}_{\pi(1)}$, then all jobs that are the sources of $\mathcal{B}_{\pi(2)}$, and so on, according to the respective IC-optimal schedules for the $\mathcal{B}_i$, and finally executing all sink jobs of $\mathcal{G}$.

Henceforth, we call the scheduling algorithm of this section *the theoretical algorithm.*

[1] That is, if $\mathcal{B}_i \rhd \mathcal{B}_j$ and $\mathcal{B}_j \rhd \mathcal{B}_i$, then the sort maintains the original relative order of $\mathcal{B}_i$ and $\mathcal{B}_j$.

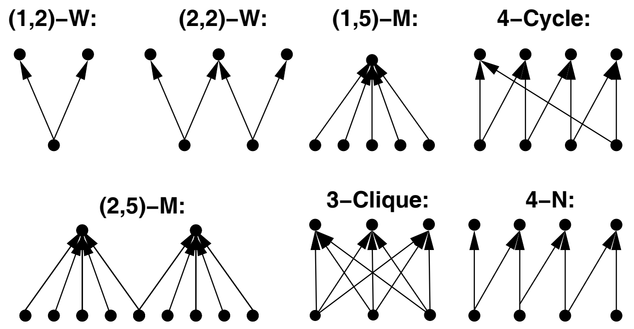


Figure 2. Sample bipartite dags with IC-optimal schedules that execute sources from left to right, then all sinks in arbitrary order.

3. A Heuristic Scheduling Algorithm and Tool

The key idea that emerges from the theoretical study [16] is that one can sometimes derive an IC-optimal schedule for a complex dag by decomposing it into simple components, scheduling each component independently, and then combining the resulting schedules. The theoretical algorithm, however, has two key shortcomings:

- The algorithm fails to find a schedule for $\mathcal{G}$ that does not have any IC-optimal schedule.
- Even for some dags that admit IC-optimal schedules, the theoretical scheduling algorithm may fail.[2]

We address these shortcomings by developing a heuristic scheduling algorithm that produces a valid schedule for every dag. The heuristic is "graceful," in that it produces an IC-optimal schedule for every dag for which the theoretical algorithm works. And, by gracefully extending

[2] Ongoing work [7, 6] is broadening the range of dags that the algorithm can schedule successfully.

the approach of the theoretical algorithm, the heuristic takes steps to enhance the production rate of eligible jobs for every dag. We then describe how we have implemented this heuristic and integrated it with the Condor system [21]. Next we list possible applications of the implementation—four scientific dags—and discuss the improvements in the rate of producing eligible jobs achieved by the implementation. We close the section by giving our algorithm engineering approaches, and sample running times of the implementation on the scientific dags.

3.1. The Scheduling Heuristic

We describe how the steps of the theoretical algorithm were modified to yield a heuristic algorithm that produces a schedule for any dag.

The Divide Phase

Step 1. This step is identical to that of the theoretical algorithm. We remove all shortcuts using an existing algorithm [13, 1].

Step 2. We generalize the decomposition process so that it never fails, i.e., so that it decomposes every dag. We accomplish this by admitting connected components that are not necessarily bipartite, at each step when no bipartite component exists.

For any source s of (the current remnant of) $\mathcal{G}'$, we let $\mathcal{C}(s)$ be the smallest subgraph of $\mathcal{G}'$ with the following properties.

1. It contains job s.
2. If $\mathcal{C}(s)$ contains a source of $\mathcal{G}'$, then it contains every child of that source.

3. If $\mathcal{C}(s)$ contains a job, then it contains every parent of the job.

$\mathcal{C}(s)$ can be constructed using a BFS-like algorithm. We start with the sets $S = \{s\}$ and $T = \emptyset$, and we iterate the following process until neither S nor T changes.

1. Add to T the children of every job in S.
2. Add to T the parents of every job in T.
3. Add to S every source of $\mathcal{G}'$ that is in T.

We decompose $\mathcal{G}'$ into dags $\mathcal{C}_1, \ldots, \mathcal{C}_n$ by iterating the following process until $\mathcal{G}'$ becomes empty. Let $s_1, \ldots, s_k$ be the sources of the current remnant of $\mathcal{G}'$.

1. Select a subgraph $\mathcal{C}(s_i)$ of $\mathcal{G}'$ that is minimal with respect to containment; i.e., choose i so that there is no j such that $\mathcal{C}(s_j)$ is a strict subgraph of $\mathcal{C}(s_i)$. One can show that any two minimal such subgraphs are either equal or disjoint.
2. Detach the selected $\mathcal{C}(s_i)$ from $\mathcal{G}'$ by removing all nonsinks of $\mathcal{C}(s_i)$ and all sinks of $\mathcal{C}(s_i)$ that are sinks of $\mathcal{G}'$.

This process produces a sequence of components $\mathcal{C}_1, \ldots, \mathcal{C}_n$ and a superdag describing how to compose the components to yield $\mathcal{G}'$.

This generalized decomposition process is equivalent to the original decomposition whenever $\mathcal{G}'$ is composed of bipartite dags. Moreover, one can show that when each component has an IC-optimal schedule, and the components can be linearly prioritized, $\mathcal{C}_1 \rhd \cdots \rhd \mathcal{C}_n$, then $\mathcal{G}$

admits an IC-optimal schedule. This motivates the subsequent steps of the heuristic.

The Recurse Phase

Step 3. We extend step 3 of the theoretical algorithm. We check if each component $\mathcal{C}_i$ is (isomorphic to) a bipartite dag with a known IC-optimal schedule. If so, we use an explicit IC-optimal schedule. If not, then we produce a schedule using a heuristic that executes jobs in the order of job-outdegree (and that thus executes sinks last), breaking ties arbitrarily. As a result, we produce a schedule for every component, and whenever a component is known to have an IC-optimal schedule, the produced schedule is IC optimal. Let each component $\mathcal{C}_i$ thereby be assigned the schedule Σ_i.

The Combine Phase

Steps 4 and 5. The verification of $\triangleright$-priorities of the theoretical algorithm is not relevant in the context of our heuristic, since we wish to provide a schedule for every dag, not just those that are assembled in a convenient way. However, we do not want to abandon the role of the $\triangleright$-priority relation as a scheduling guide, so we generalize the relation and use it in a more general way.

Let us be given dags $\mathcal{C}_i$ and $\mathcal{C}_j$, with associated schedules Σ_i and Σ_j, respectively. Say that these schedules execute sinks only after having executed all non-sinks. We say that $\mathcal{C}_i$ has *$\triangleright_r$-priority* over $\mathcal{C}_j$, denoted $\mathcal{C}_i \triangleright_r \mathcal{C}_j$, where $0 \leq r \leq 1$, when the following holds *at every step t of the computation of dags $\mathcal{C}_i$ and $\mathcal{C}_j$*. The number of eligible jobs produced by the schedule that executes all non-sinks of $\mathcal{C}_i$ according to schedule Σ_i, then executes all non-sinks of $\mathcal{C}_j$ according to schedule Σ_j, then executes all sinks in arbitrary order, is *at least* the fraction r of the largest number of eligible jobs that *could* be produced at step t by any schedule that executes both $\mathcal{C}_i$ and $\mathcal{C}_j$. Phrased mathematically, if $\mathcal{C}_i$ and $\mathcal{C}_j$ have s_i and s_j non-sinks, respectively, then:

$$\begin{aligned}
&(\forall x \in [0, s_i])\ (\forall y \in [0, s_j]) : \\
&r \cdot \Big(E_{\Sigma_i}(x) + E_{\Sigma_j}(y)\Big) \ \leq \\
&E_{\Sigma_i}(\min\{s_i, x+y\}) + E_{\Sigma_j}((x+y) - \min\{s_i, x+y\}),
\end{aligned}$$

Compare this system of inequalities with (1): when $\mathcal{C}_i$ and $\mathcal{C}_j$ are bipartite dags, and Σ_i and Σ_j are IC-optimal schedules, the $\rhd_1$-priority relation is exactly the $\rhd$-priority relation. For brevity, we omit mentioning schedules Σ_i and Σ_j when they are clear from context.

The *priority of* $\mathcal{C}_i$ *over* $\mathcal{C}_j$ is the largest r such that $\mathcal{C}_i \rhd_r \mathcal{C}_j$. Note that this quantity always lies between 0 and 1 inclusive.

Step 6. We produce a "best-efforts" schedule for $\mathcal{G}$ by greedily selecting a source of the superdag that maximizes priority. Specifically, let $\mathcal{S}$ be the superdag resulting from the decomposition of $\mathcal{G}'$. We repeat the following process until $\mathcal{S}$ is empty. Let $\mathcal{C}'_1, \ldots, \mathcal{C}'_k$ be the sources of $\mathcal{S}$. Each $\mathcal{C}'_i$ has a certain priority over any other source $\mathcal{C}'_j$. Let p_i be the smallest such priority: $p_i = \min_{1 \leq j \leq k,\ j \neq i}(\text{ priority of } \mathcal{C}'_i \text{ over } \mathcal{C}'_j)$. This says, intuitively, that, were we to execute supernode $\mathcal{C}'_i$ now, we could "lose" at most the factor $1/p_i$ of the maximum possible number of eligible jobs, over any decision. We select the next supernode to execute "greedily," by choosing a supernode $\mathcal{C}'_i$ that maximizes p_i. We schedule the non-sinks of $\mathcal{C}'_i$ according the schedule for $\mathcal{C}'_i$ constructed in step 3, and we remove $\mathcal{C}_i'$ from $\mathcal{S}$. Once having executed all non-sinks of $\mathcal{G}$ in this manner, we execute all of its sinks, in arbitrary order.

It can be shown that, when steps 4 and 5 of the theoretical algorithm succeed, the just-described greedy execution regimen achieves the same effect as does the stable sorting of a topological sort of superdag $\mathcal{S}$ mandated by step 6 of the theoretical algorithm.

3.2. The prio Tool and Condor

We implemented the scheduling heuristic of Section 3.1 and integrated it with the DAGMan (DAG Manager) component of the Condor high throughput computing system to produce a tool, called PRIO, that schedules any dag according to our heuristic.

Condor offers a CONDOR_SUBMIT_DAG command line tool for executing a dag. The tool takes a *DAGMan input file* as a parameter that specifies jobs and their dependencies. Each job has a name and an associated *job-submit description file* (JSDF, for short) determining the file that must be executed. Fig. 3 depicts an example DAGMan input file and a JSDF.

There is a mechanism to prioritize jobs in Condor. The CONDOR_SUBMIT_DAG tool has an internal queue of eligible jobs, called the DAGMan queue. Under certain conditions, jobs within this queue are forwarded to the Condor queue. The latter queue stores jobs of different users. Any user can specify the order in which her queue-resident jobs are assigned to workers, by specifying the *priority* attribute of the job specified via its JSDF. When priorities are not specified, then jobs are assigned in "FIFO order"—the order determined by the sequence in which the jobs become eligible.

The PRIO tool utilizes this mechanism. The tool:

1. takes a DAGMan input file,

2. parses the file to extract the dag of job dependencies,

3. applies the scheduling heuristic to the dag, thereby producing a schedule, called PRIO, and

4. instruments the DAGMan input file and JSDFs to assign a priority to each job.

Within the DAGMan input file, the JOBPRIORITY macro is defined for each job using the *Vars* keyword. The value of the macro reflects job's order in the PRIO schedule. Within each job's JSDF, the value of the JOBPRIORITY macro is assigned to the *priority* attribute.

Fig. 3 presents an example of invoking the PRIO tool on a DAGMan input file. The file specifies a dag composed of five jobs: a, b, c, d, and e, with dependencies $a \to b$, $c \to d$, and $c \to e$. The PRIO schedule is c,a,b,d,e. This schedule is IC optimal. The file is instrumented with five lines defining the JOBPRIORITY macro for each job; e.g., the value of the macro for job c is 5, which means that the job will have the highest priority. Each JSDF is instrumented with a single line that assigns the value of the macro to the *priority* attribute. Only one JSDF is depicted for simplicity. Jobs a and c are initially eligible. If they are forwarded to the Condor queue in a "FIFO" order, then job a is forwarded there before job c. However, the priority of job c is higher, and so Condor will then assign that job to a worker before it assigns job a.

```
FILE IV.dag                  FILE a.submit
   Job a a.submit               Executable = foo
   Job b b.submit               Log = foo.log
   Job c c.submit               Queue
   Job d d.submit               priority = $(JOBPRIORITY)
   Job e e.submit
   Parent a Child b
   Parent c Child d e
   Vars c JOBPRIORITY="5"
   Vars a JOBPRIORITY="4"
   Vars b JOBPRIORITY="3"
   Vars d JOBPRIORITY="2"
   Vars e JOBPRIORITY="1"
```

Figure 3. An invocation of PRIO on a file IV.dag results in changes (bold) to the file, and also changes to the submit description files.

We could have hard coded the value of *priority* in each JSDF, without resorting to the indirection provided by the macro, but this could lead to inconveniences, since a single JSDF may be associated with jobs of more than one DAGMan input file, and these jobs may require different priorities.

The current integration has a shortcoming. In order to enforce the order of job assignment to workers, *all* eligible jobs must be forwarded to the Condor queue, without keeping them in the DAGMan queue. If some jobs with high priorities were to be kept inside the DAGMan queue while low-priority jobs were already forwarded to the Condor queue, then Condor could assign low-priority jobs to workers, unaware that high-priority jobs are eligible. Hence, the *-maxjobs* parameter of the CONDOR_SUBMIT_DAG command that throttles the forwarding from the DAGMan queue to the Condor queue should not be used. However, Condor builds a staging file for the jobs in the Condor queue. As a result, an unacceptably large staging file may be created. That shortcoming may be alleviated by modifying Condor to enable prioritizing jobs in the DAGMan queue.

3.3. Applications

There are several computations to which PRIO could be applied. As examples, we collected four dags each representing a scientific data analysis computation comprised of many jobs:

1. **AIRSN** [12] of width 250 implements a functional Magnetic Resonance Imaging (fMRI) computation. The dag has 773 jobs.
2. **Inspiral** [9] implements a search for gravitational waves. The dag has 2,988 jobs.
3. **Montage** [3] implements an assembling of a collection of images to produce a single image of a part of the sky. The dag has 7,881 jobs.
4. **SDSS** [2] implements a search for clusters of galaxies. The dag has 48,013 jobs.

Each dag has complex job dependencies. For example, the AIRSN has a structure of a “double umbrella with fringes” (see Figure 5). There are about twenty jobs (the “handle”) that proceed a fork of width 250 (the first “cover”), followed by a join, followed by another fork of width 250 (the second “cover”), followed by the final join; each parallel job of the first fork also depends on a dedicated job (a “fringe”). The dag is actually a member of a family of AIRSN dags parameterized by width.

The other three dags also have complex structures: the Inspiral includes a non-bipartite component with over 1000 jobs, the Montage includes a bipartite component with over 1000 jobs each of whose source has from a few to about ten children some of which are shared among the sources, the SDSS includes a bipartite component with over 1,500 jobs whose each source has three children some of which are shared among the sources.

3.4. Differences in Eligibility

We exemplify the differences in the number of eligible jobs produced by the PRIO and FIFO schedules. For a given dag, let Σ_{PRIO} be the schedule produced by the PRIO tool for the dag, and Σ_{FIFO} be a schedule resulting from executing eligible jobs in the order in which they become eligible. Each schedule determines a sequence in which the jobs of the dag can be executed. Recall that $E_{\Sigma}(t)$ is the number of eligible jobs when exactly the first t jobs of the schedule Σ have been executed. We then compute, at every step t, the difference in the number of eligible jobs, $E_{\Sigma_{PRIO}}(t)-E_{\Sigma_{FIFO}}(t)$ at that step. The plots of the difference for the four dags are depicted in Fig. 4. Typically, at every step, the number of eligible jobs produced by the PRIO schedule is at least that produced by the FIFO schedule; the number is sometimes significantly higher. This suggests that the PRIO tool has the potential of generating schedules for "real" dags that keep the number of eligible jobs high.

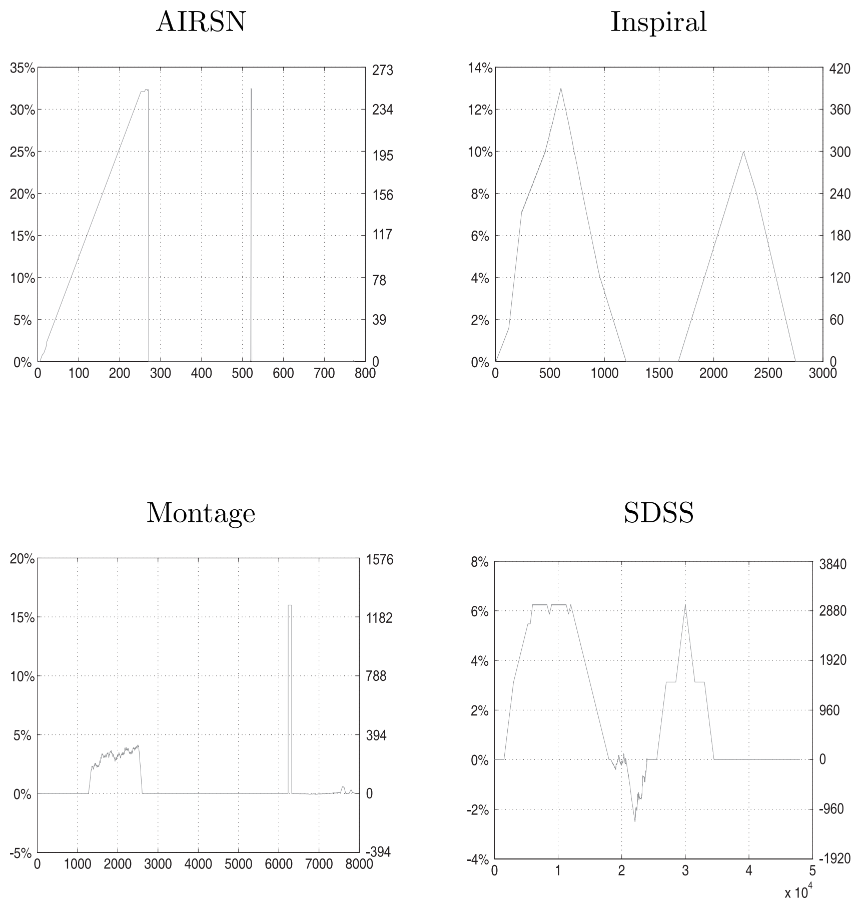


Figure 4. The differences in the numbers of eligible jobs between PRIO and FIFO schedules as a function of the number of executed jobs, normalized (left) by the number of jobs in the respective dag and absolute (right).

We can explain why the difference is so high for the AIRSN dag. Fig. 5 shows the PRIO schedule for the dag. We note that the job with priority 753 (in a black frame) has many children jobs (highlighted with a dark shade). Each of these children jobs has another parent (highlighted with a light shade). None of these (dark) children can become eligible until the (black-framed) job with priority 753 gets executed. Hence the job is in a way a bottleneck of the computation. Note that PRIO assigns high priority to that (black-framed) job and its ancestors, higher than the priority of the other (light) parents of the (dark) children. Hence when the other (light) parents get executed one by one, their (dark) children become eligible one by one, too, since the (black-framed) job with priority 753 is already executed by then. On the contrary, FIFO starts by executing the (light) parents since they are immediately eligible, before it executes the (black-framed) job with priority 753. But then when these (light) parents get executed one by one, their (dark) children do *not* become eligible pending the execution of the (black-framed) job.

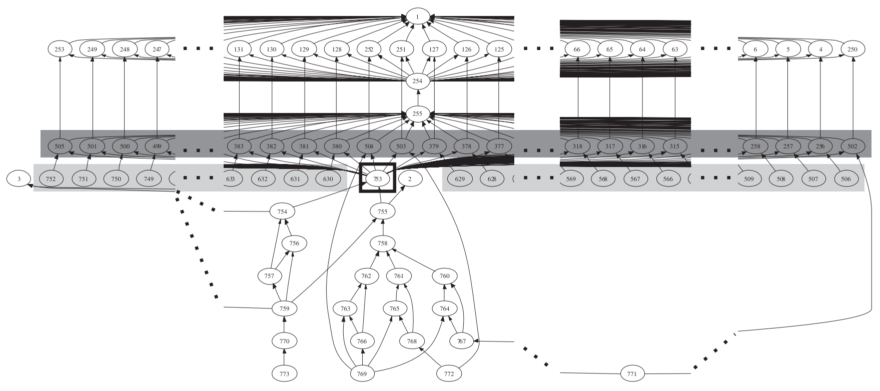


Figure 5. The AIRSN dag of width 250 with jobs prioritized by the PRIO tool; some parts of the dag are omitted. See text for details.

3.5. Improvements due to Engineering

We significantly decreased the running time of our implementation by engineering our scheduling heuristic. We note that the four scientific dags are composed almost exclusively of bipartite dags. Since containment-minimality is automatic in bipartite dags, we could accelerate our decomposition algorithm by having it first try to identify a *bipartite* subgraph whose sources are the sources of $\mathcal{G}'$, invoking a more general (and time-consuming) search for a containment-minimal subgraph $\mathcal{C}(s)$ only if there is no bipartite subgraph. This pragmatic change reduced the time to decompose the SDSS dag with 48,013 jobs from over 2 days to a few minutes. The second bottleneck in our heuristic was the processing of the superdag. There, we repeatedly select a source of the remnant superdag that has the highest minimum priority. We initially employed a naive quadratic-time algorithm, but we later replaced that with a B-Tree-based priority queue [8], which reduced the running time by a substantial factor. Our final implementation has over 8,000 LOC, or 208KB, in C++ with 11 classes. The code has been submitted for release to the Condor team.

3.6. Overhead

We report the running time and the memory consumption of the PRIO tool invoked on the four scientific dags. We compiled the tool in Microsoft Visual C++ .NET environment in release configuration and ran it on a Windows machine with 3.4GHz Pentium 4 processor and 4GB RAM. The tool was used to instrument the DAGMan input files but not the JSDFs, since the latter were not available to us. The running time and the memory consumption were:

- AIRSN with 773 jobs took under 1 second using at most 2MB of memory,

– Inspiral with 2,988 jobs took 16 seconds using at most 21MB of memory,

– Montage with 7,881 jobs took 8 seconds using at most 104MB of memory,

– SDSS with 48,013 jobs took 845 seconds using at most 1.3GB of memory.

4. Performance Gains from the PRIO Tool

We run a suite of simulations to obtain insights into the performance gains one might hope to achieve from using the PRIO tool to prioritize jobs of a given dag. As a first cut at such evaluation, we perform our simulations under the assumption that all jobs have roughly the same execution time. One might expect this assumption to be approachable in Internet-based computing environments, if the server benchmarks jobs, monitors workers' past performance and present resources (cf. [4, 14, 20]), and then matches jobs to workers accordingly. However, the assumption is certainly an idealization, since a given dag could contain a very fast job and a very slow job, and then it may be difficult to match the slow job to so fast a worker to yield the execution time similar to that of the very fast job. We nonetheless believe that the results of our simulations give insight into the potential benefits of our scheduling algorithms within real Internet-based computing environments, such as grid systems.

4.1. The System Model

We model a grid as a stochastic system. We are given a dag $\mathcal{G}$ representing a DAGMan input file with jobs and their interdependencies. We focus on executing the jobs of the dag $\mathcal{G}$ (no other dag is executed together with $\mathcal{G}$). Workers arrive at the server in batches, each worker requesting one job. The number of workers in a batch is the *size* of the batch. The size of a batch is exponentially distributed with mean μ_{BS}. The first batch arrives at time 0, and batch interarrival time is exponentially distributed with mean μ_{BIT}. The running time of a job on a worker has normal distribution with mean 1 and standard deviation 0.1.

We decided to adopt the batched model for two reasons. We explicitly batch requests allowing more than one request for jobs to arrive at a given time. The batch size follows a distribution, so as to model variability in the number of idle workers available in the grid over time. Variability exists in real grid deployments. Moreover, variability changes: sometimes variability is low, sometimes it is high. In our model, the extent of variability is parameterized by μ_{BS}. The fact that we can capture and control the extent of variability through such a simple model enables the study the influence of variability in "isolation". A more comprehensive model that explicitly models a worker temporarily quitting the computation, job prefetching, and other features of existing grid deployments is beyond the scope of this paper. Another reason for conceptually batching request is more pragmatic. One way to design a server is to make it periodically check for requests. When it checks for the kth time, many requests may have accumulated since the $(k-1)$th check. These accumulated requests correspond to a batch.

We consider two scheduling regimens. (a) An *oblivious* scheduling algorithm is specified by a total order $\mathcal{P}$ on $\mathcal{G}$'s jobs, that is used to prioritize eligible jobs. When a batch of size b arrives at the server, there is a certain number e of eligible jobs that have not yet been assigned to workers. The server selects the $\min\{b, e\}$ jobs that are the smallest under order $\mathcal{P}$, and assigns those jobs to the requesting workers. When an oblivious algorithm is instantiated with the order prescribed by the PRIO schedule (produced by the PRIO tool), we call the algorithm PRIO. (b) In contrast, a *FIFO* scheduling algorithm maintains a FIFO queue of eligible jobs. When a batch of requests arrives, jobs from the front of the queue are assigned to the requesting workers; a newly eligible job is put at the end of the queue.

A basic feature of our simulations is that workers whose requests are not filled are *not* rolled over to the next batch arrival. Our rationale is that these workers may meanwhile be "intercepted" by other computations.

We consider the PRIO and FIFO scheduling algorithms because they enable a comparison of two scheduling regimens. Currently, a widely used dag scheduler for grids, DAGMan [21], follows a FIFO order when assigning jobs of a dag to workers. The alternative method studied in this paper sequences jobs using PRIO, in an attempt to always keep the number of eligible jobs as large as possible.

We use three (related) metrics to measure the performance of an algorithm.

(1) The *expected execution time* is the mean time until all jobs of $\mathcal{G}$ have been executed. We can informally approximate this expectation by simulating the execution of $\mathcal{G}$ under our stochastic model so as to measure an execution time, and then compute the average of several measurements.

(2) The *probability of stalling* is the probability that, when a batch of requests arrives, given that there is at least one job that is unexecuted and unassigned, all eligible jobs have already been assigned. This means that the server has no new job to assign to any worker at that time, even though a job will still have to be assigned to a worker. We can informally approximate this probability in a given simulation by: (a) computing the number of batches that have arrived until the batch when the last job was assigned, (b) computing the number of these batches such that when the batch arrived, every eligible job was already assigned, and (c) computing the ratio of the latter number to the former number. We then average the ratios across several simulations.

(3) The expected *utilization* is the mean of the ratio of the number of jobs in $\mathcal{G}$, divided by the total number of requests that have arrived until the batch when the last job was assigned. In other words we measure the expectation of the quantity "satisfied/requested." We can informally approximate the expectation by: (a) computing the total number of requests in batches that have arrived in a given simulation until the batch when the last job was assigned, and (b) divide the number of jobs in the dag by the total. We then average the result across several simulations.

We compare PRIO and FIFO using these metrics. Specifically, we compute the ratio of the expected execution time of a dag $\mathcal{G}$ under the PRIO algorithm over its expected execution time under the FIFO algorithm. We compute analogous ratios for the probability of stalling and for the expected utilization.

4.2. Simulation Setup

Recall our assumption that each job runs for 1 unit of time on average. We select the mean batch interarrival time μ_{BIT} as a power of 10 in the range $10^{-3} \leftrightarrow 10^{3}$, thereby modeling both scenarios wherein workers arrive relatively frequently compared to the running time of a job ($\mu_{BIT} = 10^{-3}$) and scenarios wherein workers arrive rarely ($\mu_{BIT} = 10^{3}$). The mean batch size μ_{BS} is a power of two in the range $1 \leftrightarrow 2^{16}$, thereby encompassing the size of a large grid deployment [11, 21]. Our parameter ranges capture a wide spectrum of load conditions that grid may encounter.

Given a pair $(\mu_{_{BIT}},\ \mu_{_{BS}})$, we compute ratios of metrics as follow. We first consider execution time. Let $\mu_{_{FIFO}}$ and $\mu_{_{PRIO}}$ be the true mean execution times of the dag $\mathcal{G}$, under, respectively, the FIFO and the PRIO scheduling algorithms. Since we are unable to compute the ratio $\mu_{_{PRIO}}/\mu_{_{FIFO}}$ exactly, we determine a 95% confidence interval for the ratio. To that end, we compute an empirical sampling distribution $s_{_{PRIO}}$ of $\mu_{_{PRIO}}$, by taking p samples of the simulated execution time of $\mathcal{G}$, each being the average of q measurements. It is recommended [5] that p be chosen around 300 and q around 50 for hypothesis testing. We actually increased q to 300, in order to narrow our confidence intervals. Similarly, we compute an empirical sampling distribution s_{FIFO} of μ_{FIFO}. We use these two distributions to compute an empirical sampling distribution of the ratio μ_{PRIO}/μ_{FIFO}. Specifically, we consider every pair (x, y) of samples from s_{PRIO} and s_{FIFO} and calculate x/y, thus computing p^2 values. Whenever we encounter $y = 0$, we do not report any confidence interval. In other cases, we remove the 2.5% smallest and largest values. The resulting range of values defines a 95% confidence interval for the ratio. We also compute the mean and the standard deviation of the empirical sampling distribution of μ_{PRIO}/μ_{FIFO}. Statistics for the ratio of expected utilization and the ratio of probability of stalling are computed in a similar fashion.

4.3. Our Results and Their Explanation

We compare the PRIO and the FIFO algorithms on the four scientific dags: AIRSN of width 250, Inspiral, SDSS and Montage.

The ratios of performance metrics for the four dags are depicted in Fig. 6, 7, 8, and 9. Each figure contains three plots depicting ratios of the three performance metrics; each plot has seven sections separated by vertical lines; a section shares the same value of μ_{BIT}. Within each section, μ_{BS} increases from 2^0 to 2^{16}, left to right. Vertical segments depict 95% confidence intervals for the ratios of performance metrics; bold dots denote medians.

We describe the general trends in our results. When requests arrive quite frequently ($\mu_{BIT} = 10^{-3}$), the algorithms have about the same performance on all the dags except for SDSS. However, when the arrival is less frequent ($\mu_{BIT} \geq 10^{-1}$), PRIO has some advantage over FIFO for a certain range of batch size μ_{BS}. For a given μ_{BIT}, the advantage is maximized for a certain value of μ_{BS}. For AIRSN, that value is about 2^5; for Inspiral, about 2^9; for Montage, about 2^7; for SDSS, about 2^{13}. In particular, for AIRSN when $\mu_{BIT} = 1$ and $\mu_{BS} = 2^4$, the median of the ratio of expected execution time is below 0.85; using PRIO we obtain a gain of at least 13% in the expected execution time with 95% confidence. The gains are different for different dags—the strongest are for AIRSN and the weakest for Montage.

Intuitively, PRIO can perform better than FIFO because of increased parallelism. When there is a large number of eligible jobs, then upon arrival of a sufficiently large batch of requests, many jobs can be assigned. As a result, worker utilization can be higher and the chances that there is no work to assign at all can be lower, compared to the case when the number of eligible jobs is small. These opportunities are realized under certain conditions, however. First the dag must have a shape that allows to keep the number of eligible jobs high. For example, the AIRSN dag had a shape of an “umbrella with fringes” that enabled PRIO to keep the number of eligible jobs higher than FIFO across many steps. Second, batches should not be too big. When every batch is rather big compared to the size of the dag, then the execution time is trivially related to the length of a critical path—simply then execution

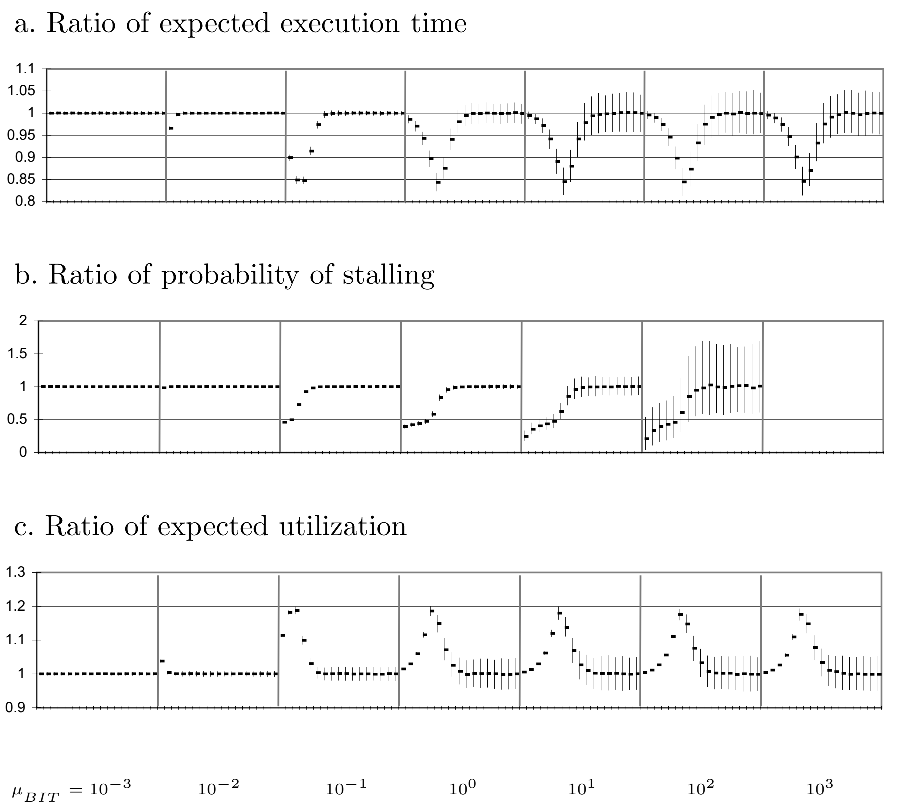


Figure 6. Performance gains for PRIO compared to FIFO for AIRSN of width 250. Line segments represent 95% confidence intervals; bold dots medians. Missing when the probability was zero.

proceeds step-by-step like a BFS traversal of the dag—irrespective of a schedule used. This is manifested by the ratios being close to 1 when μ_{BS} is close to 2^{16}. Third, an equivalent scenario occurs when batches arrive rather fast compared to how long it takes to execute a job. This is manifested by ratios being close to 1 when μ_{BIT} is 10^{-2} and 10^{-3}. Fourth, batches cannot be too small, because if they are, then the execution time is trivially related to the number of jobs in the dag—simply then execution is similar to a sequential execution on

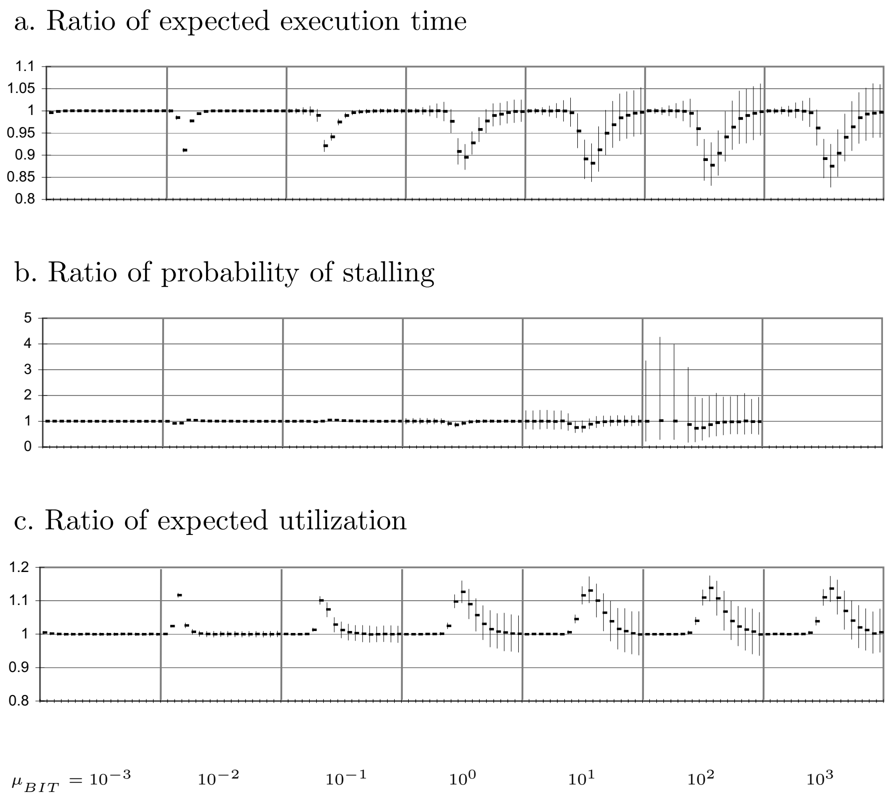


Figure 7. Performance gains for Inspiral.

one worker—irrespective of a schedule used. This is manifested by the ratios being lose to 1 when μ_{BS} is.

5. Conclusions

We have presented the design, implementation and integration with Condor of a scheduling heuristic that prioritizes jobs of any DAGMan input file. The heuristic attempts to sequence job execution so as to always keep the number of eligible jobs high, building on earlier the-

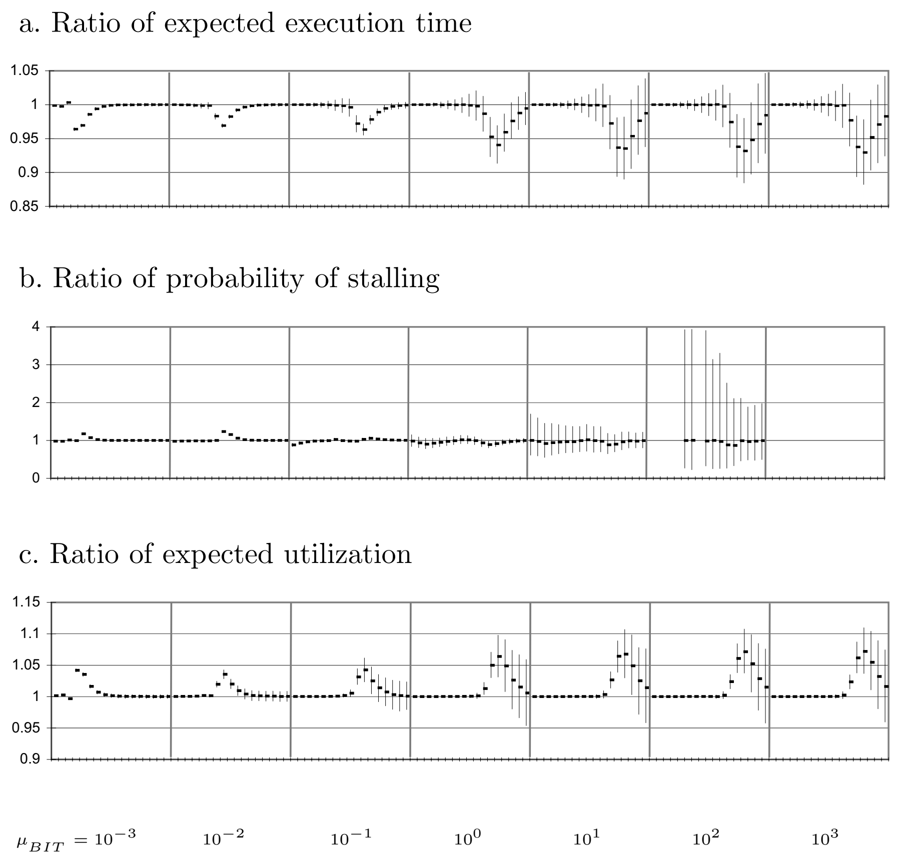


Figure 8. Performance gains for SDSS.

oretical work. We compared the heuristic with the FIFO algorithm currently used by the DAGMan component of Condor on four scientific dags. We demonstrated through a simulation that under a wide range of system parameters, with high confidence, our tool improves performance.

Our experimental results suggest that scheduling computations to maximize the number of eligible jobs is a promising optimization objective for Internet-based computing, at least when compared to FIFO scheduling. It appears from our simulations that when batches

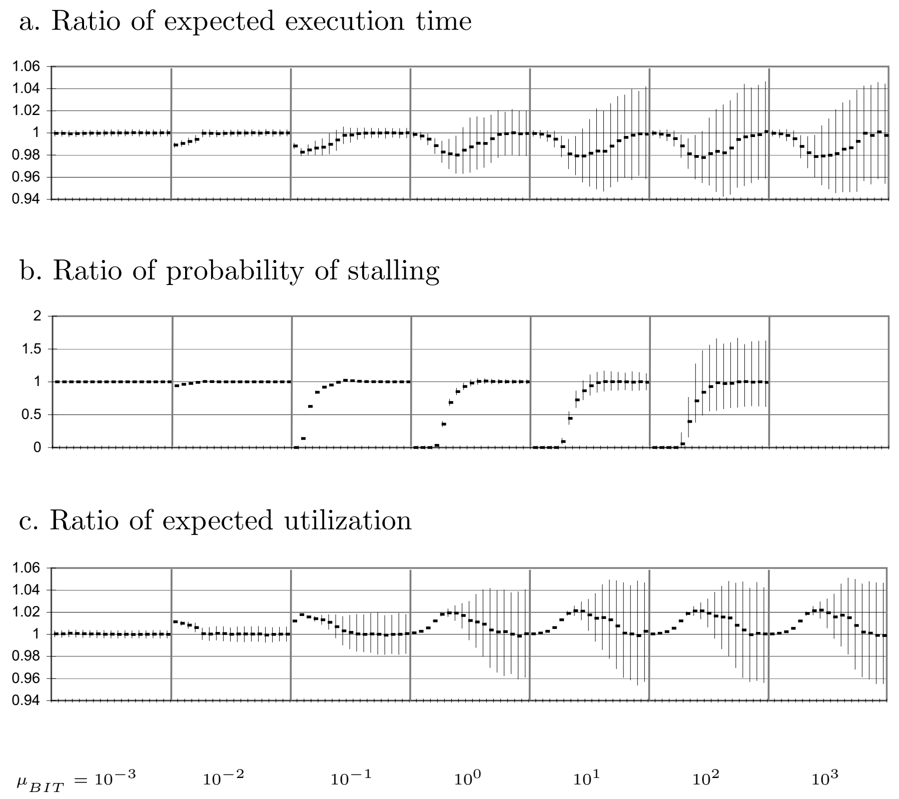


Figure 9. Performance gains for Montage.

of requests arrive rather often, or when batch sizes are either small or large, there is little difference in performance between PRIO-scheduled computations and FIFO-scheduled computations. However, when batch sizes are in the medium range, and batches arrive not quite so often—which is perhaps the more natural scenario in many Internet-based computing environments, such as grids—our simulations suggest that PRIO-scheduled computations exhibit significant performance improvements over FIFO-scheduled computations, for a wide range of system parameters. This suggests that when the relevant parameters are difficult to estimate, it may be advantageous to prioritize jobs using the PRIO tool.

More experimentation is needed to evaluate PRIO and verify the extent to which the algorithmic techniques of the underlying theory are practical. In addition to further simulations along the lines of those reported here, on a broad repertoire of other dags, it is important to test our ideas within real Internet-based computing environments. Work has already begun in both directions.

Acknowledgements

We thank James Annis (Fermi Lab), Gaurang Mehta (ISI), and Douglas A. Scheftner (U. Chicago) for the opportunity to examine scientific dags; members of the Condor team, Peter F. Couvares, Miron Livny, and R. Kent Wenger, for suggesting a method of integrating PRIO with Condor and for code review; and Argonne National Laboratory and the University of Chicago for providing access to the Jazz and Teraport clusters where simulations were performed. We are happy to acknowledge our intellectual debt to Gennaro Cordasco of U. Salerno and Matthew Yurkewych of U. Massachusetts, both collaborators in developing the theoretical underpinnings of this work. G. Malewicz wishes to thank Frederica Darema (NSF) for support that facilitated his visit to Argonne. His work was performed in part while at U. Alabama.

References

1. Aho, A.V., Garey, M.R., Ullman, J.D.: The transitive reduction of a directed graph. *SIAM J. Comput. 1* (1972) 131–137.
2. Annis, J., Zhao, Y., Voeckler, J., Wilde, M., Kent, S., Foster, I.: Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey. *ACM/IEEE Conference on Supercomputing* (2002) 1–14.
3. Berriman, B., Bergou, A., Deelman, E., Good, J., Jacob, J., Katz, D., Kesselman, C., Laity, A., Singh, G., Su, M.-H., Williams, R.: Montage: A Grid-Enabled Image Mosaic Service for the NVO. *Astronomical Data Analysis Software & Systems* (2003).
4. Buyya, R., Abramson, A., Giddy, J.: A case for economy Grid architecture for service oriented Grid computing. *10th Heterogeneous Computing Wkshp.* (2001).
5. Cohen, P.R.: *Empirical Methods for Artificial Intelligence.* MIT Press, Cambridge, MA (1995).
6. Cordasco, G., Malewicz, G., Rosenberg, A.L.: Advances in a dag-scheduling theory for Internet-based computing. Typescript, Univ. Massachusetts (2006).
7. Cordasco, G., Malewicz, G., Rosenberg, A.L.: On scheduling expansive and reductive dags for Internet-based computing. *26th International Conference on Distributed Computing Systems*, to appear (2006).
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms (2nd ed.).* MIT Press, Cambridge, MA. (2001).
9. Deelman, E., Kesselman, C., Mehta, G., Meshkat, L., Pearlman, L., Blackburn, K., Ehrens, P., Lazzarini, A., Williams, R., Koranda, S.: GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists. *High Performance Distributed Computing (HPDC'02)* (2002) 225–234.
10. Foster, I. and Kesselman, C. [eds.]: *The Grid: Blueprint for a New Computing Infrastructure (2nd Edition).* Morgan-Kaufmann, San Francisco (2004).

11. Foster, I. et al. : The Grid2003 Production Grid: Principles and Practice. *13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)* (2004) 236–245.

12. Horn, J.V., Dobson, J., Woodward, J., Wilde, M., Zhao, Y., Voeckler, J., Foster, I.: Grid-Based Computing and the Future of Neuroscience Computation. Methods in Mind, MIT Press, (2005).

13. Hsu, H.T.: An algorithm for finding a minimal equivalent graph of a digraph. *J. ACM 22* (1975) 11–16.

14. Kondo, D., Casanova, H., Wing, E., Berman, F.: Models and scheduling mechanisms for global computing applications. *16th Intl. Parallel and Distributed Processing Symp.* (2002).

15. Malewicz, G. and Rosenberg, A.L.: On batch-scheduling dags for Internet-based computing. *11th European Conf. on Parallel Processing (Euro-Par'05)* (2005) 262–271.

16. Malewicz, G., Rosenberg, A.L., Yurkewych, M.: Toward a scheduling theory for Internet-based computing. *IEEE Trans. Comput.*, to appear (2006). (Preliminary: IPDPS'05.)

17. Rosenberg, A.L.: On scheduling mesh-structured computations for Internet-based computing. *IEEE Trans. Comput. 53* (2004) 1176–1186.

18. Rosenberg A.L. and Yurkewych, M.: Guidelines for scheduling some common computation-dags for Internet-based computing. *IEEE Trans. Comput. 54* (2005) 428–438.

19. Szalay, A.S., Kunszt, P.Z., Thakar, A., Gray, J., Slutz, D., Brunner, R.J. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. *ACM SIGMOD International Conference on Management of Data* (2000) 451–462.

20. Sun, X.-H. and Wu, M.: Grid Harvest Service: a system for long-term, application-level task scheduling. *IEEE Intl. Parallel and Distributed Processing Symp.* (2003).

21. Thain D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience 17* (2005) 323–356.