Evaluating Scheduling Algorithms on Distributed Computational Grids

Ryan J. Wisnesky

Abstract. This paper presents the results of a simulation study of a heterogeneous computational grid using different scheduling algorithms. After a definition of robustness based on the concept of work completion latency is discussed, a method to simulate grids based on Estimated Time to Compute matrices is presented. Three well-known scheduling algorithms are then evaluated against each other, and the highest-performing scheduler is then analyzed in detail. The notion of ETC perturbation is presented, and this high-performing scheduling algorithm is found to be relatively robust against uncertainties in estimated task completion times.

Index Terms: grid computing, scheduling, heterogeneous systems, distributed systems

I. Introduction

COMPUTATIONAL grids are becoming more prevalent as the cost of bringing together disparate computing resources declines. However, a number of challenges remain before these grids can be utilized efficiently. This paper explores the results of using several well-known scheduling algorithms to schedule work on a grid under probabilistic work arrival rates and varying task completion times.

First, we give a definition of computational grids and argue that robustness is an important feature of economical grid computing. Then we proceed to develop a rigorous definition of robustness based on the concept of work completion latency. With this definition as our basis, we present a method for modeling computational grids and describe a software simulation framework we developed to analyze different scheduling algorithms under a variety of workloads. We then describe three scheduling algorithms and give the results of two experiments: the first to investigate the performance of the three algorithms relative to each other, and the second to investigate the effects of variation in work completion times on a specific scheduler.

A. What are Grids?

The definition of a *computational grid* is still a subject of some debate. What follows here is a short definition of a computational grid sufficient to give an adequate background for the rest of the paper; for a more in-depth definition, see [4]. Let us begin by giving an intuitive definition of what we mean by a computational grid: a computational grid is a collection of *nodes*, each of which may be thought of as a system that can perform work and has access to a network. Many systems share this property, including computer clusters. However, a grid is unique in that nodes on the grid vary in capability, and that the grid may provision more nodes to do work, or release nodes from the grid at any time.

In addition to these unique properties, the grid also has several properties stemming from its distributed nature, namely, that nodes may fail (become unable to perform work due to software or hardware problems) at any time, and communication efficiency between nodes can vary widely. In addition, nodes are typically geographically widely separated (i.e., across continents) and are typically owned by different entities. Rather than completely "owning" a node, a grid may use excess computing capacity from machines also doing other work to utilize spare computational power (for instance, the SETI@Home project uses a similar approach). A computational grid, then, can be seen as an adaptive system that provisions extra computational capacity as demand requires or as machines fail, and assigns work to nodes where the work can be done most effectively.

As promising as such a description of a full-fledged grid sounds, a number of obstacles remain before systems that have all the above properties can even be constructed, much less utilized effectively. For our purposes, we will study computational grids that are one step removed from their cluster counterparts: our grids will not expand or contract over time, nor will nodes fail, nor will there be significant communication delays between nodes; however, our grids will be made of nodes of heterogeneous capability. By studying grids of this type, we hope to enable an evolutionary approach to studying more complex grids.

B. Costs and Benefits

Why would anyone consider using a computational grid instead of a more traditional system to solve a particular problem? Simply put, grids have one main advantage over their cluster cousins: the provisioning ability of the grid enables more cost-effective solutions. The advantage stems from both the ability of a grid to use only sufficient computing capacity as demands require and the ability of a grid to use specialized hardware. Consider the case of a grid that needs to process a task involving massive amounts of vector math: the grid could provision a computer with specialized hardware to solve the problem quickly.

However, there are also certain costs to using grids. Especially in an industrial setting, there are several problems that must be overcome before grids can be adopted for widespread use. Namely:

- Given the heterogeneous and transient nature of resources on the grid, is there any way to ensure a certain level of minimum throughput, or *robustness*, against variability in the environment, like node failures?
- How is the cost of provisioning extra resources balanced against the benefit of having extra resources?

This was part of the author's CS191 project. Email: wisnesky@cs.stanford.edu

- How is amount of data that must flow between nodes minimized?
- How does the grid make decisions about where to schedule tasks?

Because we are studying grids that do not provision extra nodes, do not have nodes that fail, and have infinitely fast communication links, the question we are investigating is this: *how are tasks scheduled to a grid of heterogeneous resources to ensure a certain level of robustness?*

Ian Foster, of distributed computing fame, goes so far as to believe that a grid is not really a grid unless it is robust. In [3], one of his three requirements for a grid is that it

> *...delivers nontrivial qualities of service.* (A grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating for example response time, throughput, availability, and security, and/or co-allocation of multiple resource types to meet complex user demands, so that the utility of the combined system is significantly greater than the sum of its parts.) [Parentheses are Foster's]

Creating a robust scheduling mechanism, even for our case of limited grids, is beyond the scope of this paper. Rather, here we will discuss the performance of scheduling algorithms under assumptions of node heterogeneity, probabilistic task arrival rates, and uncertain task completion times. By investigating well-known algorithms, we can find a certain base level of robustness that specialized scheduling algorithms should be able to improve upon. This analysis, and its accompanying simulation framework, is offered as a baseline for future evolutionary steps toward constructing robust schedulers for more complex grids.

II. Problem Setup

For the rest of this paper, we may assume that a grid refers to a member of the special class of grids defined above that lacks any type of dynamic provisioning: all nodes on the grid are allocated at startup and remain that way throughout the lifetime of the grid. Furthermore, we may assume that all the communication links between nodes are reliable, and that communication latency is zero and bandwidth infinite.

Define a *path* to be an ordered sequence of *applications* $\langle A_1, \ldots, A_N \rangle$ such that the output from application A_{n-1} flows to application A_n. Intuitively, we can think of tasks as consisting of several pipeline stages, where the output from one pipeline stage flows directly into exactly one next stage, and each stage receives input from exactly one prior stage. In practice, each application has a different degree of parallelism, meaning that some applications may begin executing before all the data from the previous application is available. However, for the purposes of our simulation, we assume that there is no such overlap: a path's application A_n must finish before its next application A_{n+1} can start. An example of a task could be the multiplication of ten matrices, where each application multiplies the result from its previous application by a matrix and then feeds the result to the next application. Note that an application is the smallest unit of work that must be run completely on one node.

Define a *job* or *task* to consist of a path, an *initial data resource* (the initial data for the first application in the path), and a *customer*. The term "customer" is used to partition the set of jobs into separate classes which can then be used to develop a system performance feature. The term "customer" comes from the industry perspective of customers submitting work, but in reality any other sort of identifier could be used, or a single customer could be used. Also note that for convenience we often speak of a job's applications rather than a job's path's applications.

The grid's *scheduler* accepts jobs and distributes their constituent applications among nodes. For our purposes, we are assuming a central, omniscient scheduler that may schedule work as it becomes available; in practice, such a scheduler might be a distributed algorithm without access to all the information our scheduler has. Investigating robust, distributed schedulers is one area for future research.

A. Toward a Definition of Robustness

A definition of *robustness* makes sense only relative to a perturbation parameter. In other words, a grid might be robust against uncertainty in job completion times, or robust against uncertain work arrival rates, meaning that variations, or uncertainty, in these estimated times or changing work arrival rates do not severely affect the grid's *performance feature* (thus, robustness is necessarily grid and customer specific, because "severely" is a relative term). The performance feature is some performance metric on the grid as it is running. For instance, one interesting performance feature might be the number of jobs completed per second; another might be the amount of data processed.

1) Our Performance Feature: Latency: To evaluate robustness, we must choose our performance feature first. We proceed to give a formal definition of our performance feature for this simulation; informally, our performance feature is the percentage of jobs that "do not take too long" to finish.

Order the set of all jobs so that $job(j)$ denotes the j-th job in the ordering. Then, define $start(j,i)$ to be the time that the i-th application of $job(j)$ begins executing and define $finish(j,i)$ to be the time that the i-th application ceases executing. We will assume that

$$start(j,i) < finish(j,i)$$

and that

$$finish(j,i) \leq start(j,i+1)$$

although some realistic paths might violate the second inequality (for instance, an application that only needs the first part of the previous application's data might conceivably behave in this manner). Finally, define $finish(j,|job(j)|)$ to be the execution time of the entire job, and note that it is equal to the completion time of the last application in the job's path. Define the *latency* $L(j)$ of job j with its initial resource r_{init} to be

$$L(j) = finish(j,|job(j)|) - arrival(r_{init})$$

where $arrival(r_{init})$ denotes the time that the initial resource is first available. Let $L_c^{max}(j)$ denote the maximum acceptable latency $L(j)$ of $job(j)$ for customer c (this number, specified by the customer, simply denotes how long the customer is willing to wait for this job to complete. This notion of acceptable latency is a key component of the performance feature: for modeling, we can assume it to be a parameter, but in practice, this number would likely be defined by the customer as part of a service level agreement with the company providing the grid). Then, we define $J_c(t)$ to be the set of all jobs that arrive at time t for customer c. We also define the helper function

$$1(x, y) \equiv \begin{cases} 1 & \text{if } x > y \\ 0 & \text{otherwise.} \end{cases}$$

Now, we define $\alpha_c(t)$ to be the *job latency failure rate*, meaning that

$$\alpha_c(t) = \sum_{\forall j \in J_c(t)} 1\big(L(j), L_c^{max}(j)\big)$$

Intuitively, $\alpha_c(t)$ is the number of jobs, submitted by customer c at time t, that are taking too long to finish. The *performance feature* $f(T)$ of the grid over a discrete interval T is then simply

$$f(T) = \sum_{\forall c} \sum_{\forall \tau \in T} \alpha_c(\tau)$$

Intuitively, the performance feature of a grid is the number of jobs missing their acceptable latencies. Generally, we will divide $f(T)$ by the number of jobs arriving during the interval T to obtain the performance as a more useful percentage.

2) Adequate Performance: Before we can define what we mean for our grids to be robust against variations in a perturbation parameter, we must first define what it means for the grid to be *performing adequately*. We will then say that the grid is robust when it still performs adequately when probabilistic variations in the environment, expressed through the perturbation parameter, are introduced.

Let the function $\beta_c^{max}(t)$ be the *maximum job latency failure rate* for customer c for the jobs submitted at time t. This number, specified by each customer, indicates the number of jobs they are willing to accept taking longer than their maximum acceptable latency at a given time. In practice, this will probably be specified as a percentage, but without loss of generality we can assume it to be an integer. This quantity necessarily varies with time because it could be the case that a customer is willing to accept more failures at certain times than others. The *adequate performance metric* $\delta_c(t)$ for customer c is then defined as

$$\delta_c(t) = \beta_c^{max}(t) - \alpha_c(t)$$

The system is *performing adequately* when for every customer c, and at every moment of time t, $\delta_c(t)$ is positive. Intuitively, a system is performing adequately when, at every moment, each customer is satisfied that not more than some given percentage of their jobs are taking too long.

Of course, this definition of adequate performance is a boolean condition: either the system is adequately performing or it is not. We can use the adequate performance metric $\delta_c(t)$ to create other, more nuanced definitions of adequate performance; a number of examples come readily to mind. We could say that a system is maximally adequately performing when $\delta_c(t)$ is always the maximum possible over any t, or that a system is averagely adequately performing when the average value of $\delta_c(t)$ over the system lifetime is greater than zero. Investigating applications of the adequate performance metric and associated performance features is one of the important directions for future work.

Note that for our simulation, we report the performance features for grids rather than their adequate performance. We do this because the notion of adequate performance is tied to the customer-specified maximum job latency failure rate; the definition of the performance feature is not. This definition of adequate performance is offered as the basis for a definition of adequate performance that might be used in practice as the foundation of a service level agreement by the grid provider and customer.

3) Our Robustness: As stated earlier, a system must be robust relative to some perturbation parameter. There are many such parameters, including

- node failures
- variations in job arrival rates
- variations in estimated job completion times
- time (i.e., a robust system's performance does not vary with time)

We have chosen variation in job completion times as our perturbation parameter for this simulation. The simulation framework itself also could support an evaluation of robustness against job arrival rates; this is another area for future work.

B. Calculation of ETC Values

In order to evaluate the robustness of a scheduling algorithm against variation in estimated job completion times, we must first find a way to model these completion times. We use the method described in [5] to model the application execution times for our simulation. A brief synopsis of the relevant points, along with our modifications to the method, is described in this section.

1) ETC Matrices: The term *ETC value* is short-hand for "**E**stimated **T**ime to **C**ompute value." An ETC value represents the amount of time that an application needs to run on a given node in order to complete – every ETC value must be relative to some node; there is no notion of how long an application takes in the abstract. An *ETC matrix* is a matrix that has an ETC value for every application (the rows) and node (the columns). This definition can best be described by an example; see Table I. Note that because of the requirement that every entry in the matrix have a value, we are assuming that every application can be run on every node. Extending this method to take into account applications that may only run on certain nodes is an interesting extension.

TABLE I
AN EXAMPLE ETC MATRIX FOR A PATH WITH 7 APPLICATIONS ON A GRID OF 5 NODES

	n_1	n_2	n_3	n_4	n_5
a_1	21	7	4	5	9
a_2	6	14	11	18	9
a_3	17	13	13	15	11
a_4	8	19	12	16	15
a_5	13	5	9	10	7
a_6	10	12	13	21	5
a_7	4	15	17	11	9

2) Creating ETC Matrices: [5] gives a generous development of the procedure we use to generate ETC matrices. In essence, the procedure used to create an ETC matrix for a given path takes as input parameters

Mutask measures the central tendency of the ETC values for applications on “average” nodes
Vtask measures the variation of ETC values from Mutask
Vmach measures node heterogeneity

The procedure then uses the parameters to create several gamma distributions, which are then sampled to create the values in the matrix. The algorithm is given in Figure 1.

Fig. 1. The CVB Method for generating ETC Matrices. Summarized from [5].

```
Inputs: Vtask, Vmach, Mutask
Outputs: e[][] // the ETC matrix
Begin:
  Let Atask = 1/(Vtask*Vtask)
  Let Amach = 1/(Vmach*Vmach)
  Let Btask = Mutask/Atask
    For each task i:
      Let q = Gamma(Atask, Btask)
      Let Bmach = q/Amach
      For each node j:
        e[i,j] = Gamma(Amach, Bmach)
```

This method is capable of generating ETC matrices that have high-task, high-machine heterogeneity; high-task, low-machine heterogeneity; and low-task, low-machine heterogeneity. Unfortunately, this method cannot generate low-task, high-machine heterogeneity matrices. A related algorithm can generate exactly those cases that the original algorithm cannot supply. However, for our simulation, we will be using medium-task, medium-machine heterogeneity matrices.

Unfortunately, this method does not generate *consistent* ETC matrices. A consistent matrix has the property that *if a node n has a lower ETC value than a node m for any task, then the same is true for all tasks*. This is a desirable property is many situations; as the authors of [5] discuss, an Intel Pentium III would likely be faster than an Intel 286 for every conceivable task. There is a method for extracting consistent matrices from inconsistent matrices; however, because the generated matrices tend to be *partially consistent*, meaning that they have at least one consistent sub-matrix, and because inconsistent and partially-consistent matrices tend to occur in environments where tasks have vastly different computational needs (i.e., some are floating-point intensive, others need multiple threads, etc.), we feel that using inconsistent matrices is justified.

Our extension of the method given in [5] is to simply apply the method to generate application ETC values for paths. Thus, a path *is* an ETC matrix.

C. Scheduling

The grid scheduler assigns applications to nodes. Each node can run one application at a time, and must run that application to completion. (By splitting multi-cpu machines into a set of single-cpu nodes, powerful machines may in fact run more than one job at a time.) The scheduler maintains a queue of work for each node, and may re-order the queue at any time. An application that is being run but whose previous application has not finished will wait until the previous application has finished before starting. (This is a simplifying assumption: in reality, some applications may begin before their previous application has finished. For instance, a sequence of applications that are streaming data to each other has this property.)

The scheduler is invoked when *mapping events* occur. During a mapping event, the scheduler may reorder each node’s work queue (thereby *mapping* applications to nodes), and assigns new jobs that have arrived to various nodes. In general, mapping events occur whenever

- a new job arrives
- a node enters or exits the grid
- an application finishes or is aborted
- acceptable robustness/performance changes

However, for the purposes of this simulation, mapping events are only fired when new jobs arrive, and when applications finish. In general, a robust scheduler will need to take robustness into account when making mapping decisions. It is interesting to ask, however, how well scheduling algorithms which are not aware of robustness will perform. We have selected the following three scheduling algorithms to study:

- *FCFS*: First come, first serve. Maintains a queue of applications, in the order they arrive at the grid, and assigns them to the nodes in the order the nodes become available. When an application finishes running, if there is another application to run to complete that application’s associated job, then that application will be added to the end of the incoming work queue. Thus, this is a FCFS algorithm on applications, not jobs.
- *RR*: Round-Robin. Runs all the applications from the first job on the first node, all the applications from the second job on the second node, and so on. It is also possible to round robin applications, rather than paths, to nodes, but path-based round robin was chosen for simplicity.
- *MCT*: Minimize Completion Time. Assigns each application in a job to the set of nodes that will spend the least time performing computation (i.e., the set of nodes with the shortest latency, but not necessarily the set of nodes that will produce the result first). This is achieved by keeping a priority queue for each node that holds its work to be done, ordered such that jobs that entered the grid earlier are given higher priority. When an application finishes running, the next application that must run to complete the job is placed at the appropriate spot in the appropriate queue. *Note that this is the only scheduler that uses ETC values when making scheduling decisions*. Also note that a priority queue is used because although a regular queue would also minimize completion time, it would do so without regard to latency. In other words, our heuristic is that to help minimize latency, older applications should be run before new applications, which is implemented by the priority queue. This heuristic does not hold in all cases, but it is a good rule of thumb that makes for a more realistic (and interesting) scheduler.

Each of these schedulers assumes perfect knowledge of the grid. This means that, when a node completes a task, the scheduler is notified; the scheduler itself maintains a queue of work for each node. In practice, with a grid distributed over continents and with large communication latencies between nodes, implementing a centralized, omniscient scheduler might be impractical. However, it is useful to know how an idealized scheduler performs in simulation before attempting to implement any scheduler in practice.

III. Simulation Description

A. Overview

Our goals for the simulation were two-fold. First, we measured the performance features of each of the three schedulers, under a variety of conditions, to get a general idea of how the schedulers perform. Then, we introduced variations in the actual completion times for tasks and measured the robustness of the MCT scheduler to these perturbations in ETC values. (We only examined MCT under perturbation because RR and FCFS do not use ETC values. Although perturbing task completion times for these two schedulers would likely cause variations in robustness from simulation to simulation, the changes in robustness are uninteresting: because ETC values are not used when making scheduling decisions, a perturbed system is essentially equivalent to another system with different ETC matrices; this is not the case in an MCT system.)

B. General Framework

The simulation framework, written in Java, is relatively simple. The simulation runs a series of *ticks*, simulating the passage of time. A `Scheduler` is notified about mapping events and assigns applications to nodes. The `Application`s then spin repeatedly as ticks occur, notifying the framework when they are complete. Each `Job` is given an ETC matrix using the method described above.

C. Parameters

Because it is computationally expensive to generate ETC matrices for hundreds of jobs, the simulation framework creates a number of *template jobs*. Each template job has an associated *arrival rate*, which determines the average number of instances of that job that enter the simulation per tick.

In order to simulate realistic environments, we must decide on values of Vtask, Vmach, and Mutask for each template job so that we can generate ETC matrices by the method in Figure 1. (Of course, “realistic” is a relative term; our grids are already highly idealized. By realistic ETC values, we simply mean ETC values that might conceivably be encountered in practice.) To do this, we created the following initial values from which we deviate randomly as we create template jobs (how we do so will be described afterward):

- vmachinit = .4 (The central tendency of Vmach)
- vtaskinit = .4 (The central tendency of Vtask)
- mutaskinit = 20 (The central tendency of Mutask)
- vtaskvar = .2 (Variation for vtaskinit)
- mutaskvar = .1 * mutask (Variation for mutaskinit (2 ticks))
- numapps = 10 (The central tendency for number of apps per job)
- numappsvar = 7 (Variation for number of applications per job)
- ma = .01 (The central tendency for one job arrival per tick)
- mavar = .005 (Variation for ma (50%))

For each template job, we use the following algorithm to generate the three parameters for its ETC matrix (the Gaussian function samples a Gaussian curve over $(-1..1)$, and the uniform random function samples over the interval in the call’s parameters):

```
vtask = nextGaussian()*vtaskvar+vtaskinit
mutask = nextGaussian()*mutaskvar+mutaskinit
numtasks = numapps +
 uniformRandom(-numappsvar, numappsvar)
```

The acceptable latency for each template job, and thus for each job instance of that template, is defined to be 1.5 * Mutask * numapps. (Note that as different template jobs have different Mutask and numapps values, not every job has the same acceptable latency.) This choice of acceptable latency is arbitrary, as in practice it would be specified by the customer; this value seems like a reasonable one. Also note that by defining the acceptable latency this way, we are essentially simulating a system with a single customer. Finally, note that we simulating a system that has large numbers of a few types of identical jobs. An example of such a system might be encountered on a grid that has to process thousands of database reads and writes.

In addition, we say that for each template job, its arrival rate is a Poisson function with mean

```
mean = nextGaussian() * mavar + ma
```

Note that each template job has its own arrival function; as the number of template jobs is increased, the system will become more loaded. For all the simulation runs, we use 20 template jobs.

D. Simulation Body

The body of the simulation is as follows:

1) Create the 20 template jobs and associated ETC matrices.
2) For each tick, do the next 3 steps:
    a) For each template job, query its arrival function to determine if an instance of it should arrive this tick.
    b) If a job needs to arrive, clone an instance of the job from the template and notify the scheduler.
    c) If applications have finished, notify the scheduler.
3) Finally, at the end of 10,000 ticks, count the number of jobs that have finished within their acceptable latency, those that have violated that latency, and those still in progress. 10,000 ticks were chosen to give the system enough time to enter a steady-state and to simulate long running grids.

Each simulation run is given a specified number of nodes; we vary the number of nodes while keeping the workload constant and measure performance. An average job has around seven applications that each take around 20 ticks; therefore a steady-state will certainly have been reached by 10,000 ticks.

E. How Realistic is this Environment?

Given our already highly idealized conception of grids (no node failures, no communication latency, etc), it is natural to ask how realistic this environment is. If we look at the sample ETC matrix in Figure 1, we see that the times are not implausible for certain situations. Especially on grids where nodes are running other applications concurrently (i.e. the node is not “owned;” the grid is using “spare cycles”), large variations in ETC values would be expected. The issue of communication link latency is more difficult. Presumably, for certain grids and applications, we can simulate communication link latency by adding appropriate extra time to a job’s ETC values. For other grids, allowing each application to pull data as soon as it is available from its predecessor application could hide communication latency. Developing an abstraction to allow use of ETC matrices with parallelizable jobs, and with varying communication latency and bandwidth, is an important direction for future research. However, we feel that this environment is still worth investigating, especially given the youth of the field.

IV. Results

A. Results for Adequate Performance

1) Preliminaries: Each run of the simulation consists of ten runs of the simulation body above. The statistics are totaled across the loop bodies, but the template applications are regenerated and the simulation reset during each execution of the loop body. The results are summarized in Tables 2-5. “OK,” “Late,” and “IP” refer to the number of jobs that finished within their acceptable latencies, finished outside their acceptable latencies, and did not finish as of 10,000 ticks, respectively. Because the performance feature is meant to capture the steady-state of the system, jobs in-progress at the end of 10,000 ticks were ignored while calculating performance. The performance is the percentage of jobs that finished within their acceptable latency, as defined earlier.

2) Data: See Figures 2-4 and Tables II-IV.

TABLE II
FCFS Results

Nodes	OK	Late	IP	Perf (%)
1	0	0	1820	0.000
2	0	0	1793	0.000
4	2	24	1762	0.077
8	15	93	1672	0.139
16	108	314	1415	0.256
20	271	394	1194	0.408
24	504	464	829	0.521
28	866	346	674	0.715
32	1397	6	362	0.996
64	1557	0	323	1.000

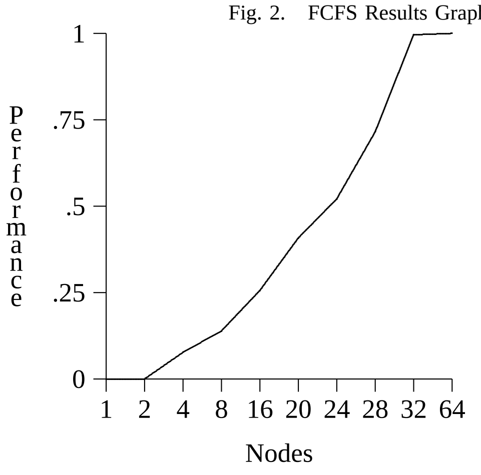
TABLE III
RR Results

Nodes	OK	Late	IP	Perf (%)
1	4	15	1810	0.211
2	25	73	1755	0.255
4	72	179	1560	0.287
8	134	271	1380	0.331
16	375	432	1046	0.465
20	576	413	874	0.582
24	837	350	682	0.705
28	1029	310	574	0.768
32	1256	116	442	0.915
64	1511	0	327	1.000

TABLE IV
MCT Results

Nodes	OK	Late	IP	Perf (%)
1	2	14	1739	0.125
2	26	91	1647	0.222
4	103	262	1416	0.282
8	401	405	1013	0.496
16	1471	62	351	0.960
20	1648	0	134	1.000
24	1697	0	120	1.000
28	1653	0	95	1.000
32	1756	0	107	1.000
64	1799	0	63	1.000

Fig. 2. FCFS Results Graph



3) Discussion: When interpreting the data, it is important to remember that achieving a steady-state has nothing to do with the performance features. For instance, for FCFS with small number of nodes, very few jobs are finishing; this is because the scheduler is assigning applications to nodes in a way that precludes jobs from finishing. This is, in fact, a steady-state, where there simply aren’t enough nodes available to finish jobs quickly enough.

We see that both FCFS and RR have similar performance; both algorithms required a large number of nodes before maximum performance was achieved. MCT attained maximum performance extremely quickly, as we would expect for an algorithm that schedules jobs to minimize completion time.

It is interesting to note that FCFS consistently tended to perform slightly more poorly than RR. This can be explained by remembering that FCFS, when an application completes, will add the next application in the associated job to the end of the work queue, thereby requiring that application to wait for some time before starting. In effect, FCFS adds a long delay between applications in a job when the system is overburdened. Both RR and FCFS achieved a steady state maximum performance at around 30 nodes, however, because both FCFS and RR tend to execute applications on average nodes (that is, FCFS will assign essentially a random node to an application, and RR will assign a different random node). Because both assignments are random and average, we would expect maximum performance to be achieved in roughly the same spot.

Fig. 3. RR Results Graph

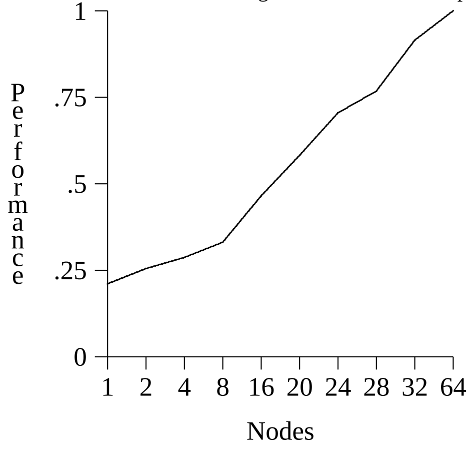
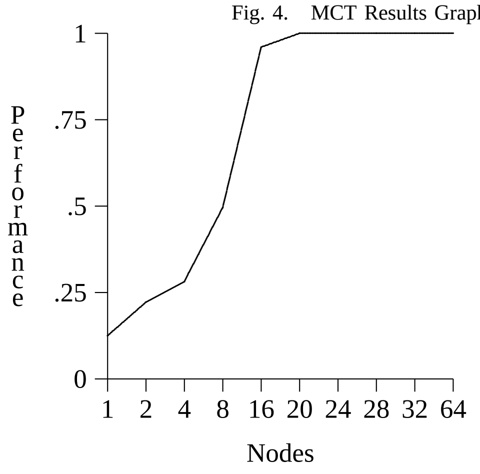


Fig. 4. MCT Results Graph



B. Results for Robustness

1) Preliminaries: In a sense, the above analysis simply confirmed what we might have expected: MCT performs much better than the other algorithms. We saw that MCT achieves perfect adequate performance for our environment when it has around 16 nodes. It is then natural to ask what MCT's robustness against variations/perturbations in ETC values is. In other words, how does MCT perform when the ETC values it is given no longer match the actual time it takes tasks to complete?

To introduce uncertainty into ETC values, we calculate ETC matrices as usual. Then, when a job begins running, we assign it a different time to completion based on a parameter `etcvar`:

```
actualetc = ((1-etcvar)*(etcvar)
            * nextGaussian(-oldetc, oldetc)
            + oldetc
            + (etcvar)*nextUniform(-oldetc,
            oldetc) + oldetc)/2
```

Thus, the scheduler assigns jobs to nodes based on *actual estimates* of completion times; before, when ETC values were the *real* completion times, the ETC values were perfect estimates. Now the ETC values truly do estimate uncertain completion times.

The intuition behind the formula is straightforward: in a system that is unperturbed (`etcvar=0`), the ETC values will be true estimates of job completion times; therefore, the actual distribution of job completion times for that job template should be a Gaussian centered at the ETC value. In a completely perturbed system (`etcvar=1`), the ETC value is the center of a uniform distribution whose average is the ETC value. For systems in between these extremes, a weighted average of the two values is used. We define this variation from a Gaussian distribution to be the *ETC perturbation*. Figures 5 and 6 demonstrate high and low perturbation.

Other definitions of perturbation are possible. For instance, rather than using uniform and Gaussian distributions, other distributions could be used; or, perturbation could correspond to the correlation of ETC values and the actual samples from the distribution. Our definition is offered as a simple, intuitive starting point.

Fig. 5. Low Perturbation

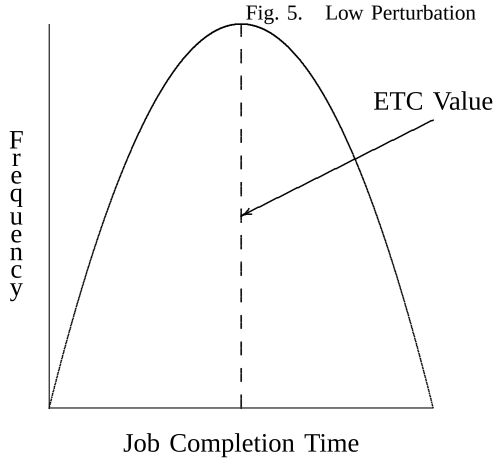
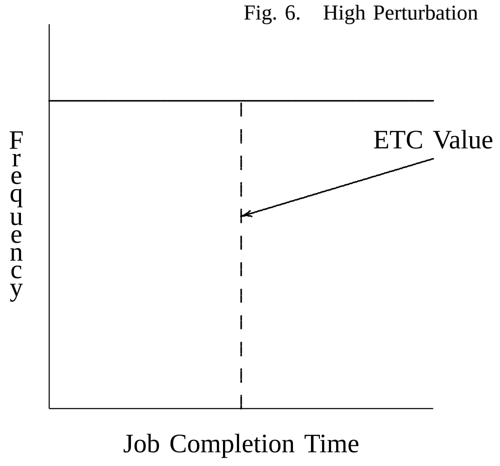


Fig. 6. High Perturbation



To determine how robust MCT is against ETC variation, we use `etcvar` as the perturbation parameter and examine adequate performance as above. We vary `etcvar` is increments of .1 (this increment is arbitrary; it is a compromise between too few and too many data points). Each simulation runs for 10,000 ticks, as before; however, these simulations are run three times and then their results averaged. For the sake of space, all the results are not included here.

2) Data: The data are available in Table V.

TABLE V
MCT VARIATION RESULTS

nodes	0	.2	.4	.6	.8	1	Max - Min
1	.13	.19	.12	.20	.20	.21	.15
2	.21	.24	.21	.21	.19	.20	.05
4	.21	.29	.29	.31	.30	.31	.10
8	.47	.44	.47	.46	.49	.43	.06
16	.94	.96	.96	.97	.97	.95	.04
20	1.0	1.0	1.0	1.0	1.0	1.0	0.0

3) Discussion: We see that MCT is remarkably stable against perturbation in ETC values. In fact, all degrees of perturbation seemed to make very little difference. However, performance values did tend to be slightly more diffuse than in the first experiment, hence the need to repeat the trials more than once. This stability suggests a more general result: as long as ETC values represent the average actual completion times of the associated jobs, then the actual underlying distribution of the job completion times does not matter to MCT. This result could potentially be generalized to any scheduler, indicating that ETC matrices might be useful as inputs to actual robust schedulers, rather than only being useful as tools for a simulation. This possibility would drastically increase the utility of using ETC values for scheduling decisions, as it demonstrates that a concept as simple as an ETC value could be useful in practice. A mathematical proof of this result for MCT is one direction for future work.

Of course, caution is urged. It is impossible to tell, from this experiment alone, whether this result is general or if it is the result of this particular system and its associated ETC matrices. For instance, we cannot know whether this stability will change if jobs are inserted at a slightly faster rate; we also cannot know whether this result holds for any scheduler besides MCT. The stability of MCT might be related to the MCT algorithm itself rather than ETC values being averages of actual completion times. Verifying this result for other environments and with other schedulers is yet another direction for future work.

In addition, even if ETC matrices do turn out to be useful in practice, effectively calculating the ETC matrices for a given grid with a given workload is still an open problem. Approximations created with the described method would be an excellent first step, but the problem is not trivial, and should prove to be an interesting area to investigate. Scheduling methods that do not require estimates of completion times are investigated in [2].

V. Conclusion

We have presented the results of a simulation study of a heterogeneous computational grid. After discussing a definition of robustness based on work completion latency and presenting a method using ETC matrices to model heterogeneous systems, we found that the MCT scheduling algorithm – an algorithm which attempts to minimize the total computational time required for any job – performed the best out of a set of well-known scheduling algorithms. To analyze MCT further, we presented the concept of ETC perturbation, and found that the MCT algorithm was quite stable against variations in job completion times.

However, the most interesting result of the simulations was not the data itself, but the intriguing possibility that ETC values might be useful as inputs to robust schedulers. That is, it could be the case that ETC matrices, even when they are only rough estimates of underlying job completion times, could be treated as true estimates without affecting robustness. Although more work is required to generalize our result, if ETC matrices could be used in this way, then we are one step closer to creating robust scheduling algorithms.

VI. Related Work

The concept of computational grids, and grid computing in general, is being studied by researchers in many fields, including high-performance computing, networking, distributed systems, and web services. [4] is an extensive introduction to what a computational grid actually consists of, and what is required to implement it. The Globus Consortium (`http://www.globus.org`) is a consortium of dozens of companies, government agencies, and universities that is creating an open standard for grid development using web-services as an RPC mechanism.

The modeling of computational grids with heterogeneous resources is just beginning to be explored. [5], published in 2002, can point to no directly related work in the field.

Scheduling tasks of unknown duration on distributed systems is investigated in [2].

The evaluation of scheduling algorithms focused on efficiency rather than robustness is explored in [1].

The construction of actual grids for industrial and scientific work has been undertaken by many companies and scientific groups. One particular success story is the Grid 2003 project (`http://www.ivdgl.org/grid2003`), which has developed a grid consisting of 2000 CPUs spread across the world.

The development of robust scheduling mechanisms is being investigated by IBM. Jay Smith (`bigfun@us.ibm.com`) is one contact point for this work.

Acknowledgements

I would like to thank Jay Smith of IBM for his help in creating the simulation framework and writing this paper. His investigations of ETC matrix methods and modeling techniques proved invaluable, and his definition of robustness was central to this research.

I would also like to thank Professor Mendel Rosenblum for graciously agreeing to supervise my work this quarter.

References

[1] C. Boeres et al. *A tool for the Design and Evaluation of Hybrid Scheduling Algorithms for Computational Grids*. Proceedings of the 2nd workshop on Middleware for grid computing, ACM International Conference, October 2004.

[2] Harchol-Balter, Mor. *Task Assignment with Unknown Duration*. Journal of the ACM, Volume 49 Issue 2, March 2002.

[3] Ian Foster. *What is the Grid? A Three Point Checklist* Grid Today, volume 1 number 6, July 22 2002.

[4] Ian Foster, Carl Kesselman. *Computational Grids*. Chapter 2 of *The Grid: Blueprint for a Future Computing Infrastructure*. Ian Foster and Carl Kesselman, Morgan Kaufman, 1998.

[5] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, Debra A. Hensgen, Sahra Ali. *Task Execution Time Modeling for Heterogeneous Computing Systems*. Heterogeneous Computing Workshop, 2000: 185-199.