IDEA GROUP PUBLISHING ITJ3530

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

This paper appears in the publication, **International Journal of Information Technology and Web Engineering**, Volume 2, Issue 1 edited by **David C. Rine © 2007, Idea Group Inc.**

Hierarchical Scheduling in Heterogeneous Grid Systems

Khaldoon Al-Zoubi, Carleton University, Canada

ABSTRACT

This article proposes hierarchal scheduling schemes for grid systems: A self-discovery scheme for the resource discovery stage and an adaptive child scheduling method for the resource selection stage. In addition, we propose three rescheduling algorithms: (1) The butterfly algorithm, which reschedules jobs when better resources become available, (2) the fallback algorithm, which reschedules jobs that had their resources taken away from the grid, before the actual resource allocation, and (3) the load-balance algorithm, which balances the load among resources. We also propose a hybrid system to combine the proposed hierarchal schemes with the well-known peer-to-peer (P2P) principle. We compare the performance of the proposed schemes against the P2P-based grid systems through simulation with respect to a set of predefined metrics.

Keywords: grid computing; grid environment; grid systems; hierarchal scheduling; P2P systems; cluster computing; parallel scheduling; high-performance computing; parallel processing

INTRODUCTION

The current status of computation is equivalent in some respects to the status of electricity circa 1910s (Foster & Kesselman, 2004). At that time, electrical power was generated by generators for specific individuals or organizational needs. Truly, the real influence of electricity in our lives was born with the creation of the electric power grid, which was provided via sharing generators. The "grid computing" term was adopted from the electricity grid to *amplify computational power* via *sharing* computational resources, since both grids are similar with respect to their infrastructure and purpose. The term "the grid" started in the mid-1990s (Foster, 2001; Foster & Kesselman, 2004) to portray the infrastructure of both scientific and commercial computing communities and has been gaining popularity ever since. The "grid" can be defined as a parallel and distributed system that enables a large collection of geographically distributed heterogeneous systems that usually span several organizations to share a variety of resources dynamically, depending on their availability, capability, user's requirements, and any other predefined rules set by local systems and resources owners. The type of sharing in the grid gives the impression of a powerful self-managing virtual computer. The Internet is an ideal choice to link thousands or millions of computers, since it already connects the whole world—if a node's IP address is known, it can then receive data from another node. Benefits of grids can be extensive. They include: (1) expanding computing power, since grids unleash the hidden computing power that is not being used most of the time (e.g., most machines in a typical organization are busy less than 5% of the time (Berstis, 2002)), (2) improving productivity and collaboration among organizations (i.e., wider audience) in a dynamic and geographically distributed manner to form one powerful computing system, and (3) solving complex problems that were previously unsolvable.

The rest of the article is organized as follows. In the next section, grid scheduling stages and some of the grid challenges are described. Then, the self-discovery method is presented. It is used in the resource discovery stage and the adaptive hierarchical scheduling (AHS) method, which is used in scheduling jobs on selected resources. Note that the AHS method is based on the AHS method for parallel and cluster systems presented in Dandamudi (2003). In addition, we present three rescheduling dynamic algorithms: the butterfly, the fallback, and the load-balance. Next the simulation model and samples of the results are given. Refer to Al-Zoubi (2006) for more a more in-depth discussion of the presented schemes and the complete set of results. The results are followed by conclusions.

GRID SCHEDULING STAGES

Grid characteristics must be taken into account in order to perform efficient scheduling. Grid schedulers must make scheduling decisions in a very challenging environment that includes: (1) no control over the resources, since they don't own them; (2) distributed resources; (3) a dynamic existence of resources (i.e., resources may be added or removed from the grid at any time); (4) a dynamic information collection; (5) heterogeneous resources (jobs must match appropriate resources in order to be executed as requested by the users); and (6) tentative scheduling until the allocation of actual resources (i.e., resources may be taken from the grid before a job actually uses them).

In general, grid scheduling is performed in three stages (Nabrzyski, Schopf, & Weglarz, 2004). First is the resource discovery stage, which produces a set of matched resources. Schedulers are expected to collect static information (e.g., operating systems) from local schedulers or general information systems (GIS), in order to perform job matching. In the next stage, resources are selected (i.e., resource selection stage) from the list obtained during the first stage and are expected to meet user's imposed constraints (e.g., deadlines). Then schedulers are expected to collect dynamic information (e.g., system load) for the third stage and transfer jobs to selected resources (i.e., job execution stage).

HIERARCHAL SCHEDULING IN GRID SYSTEMS

Grid schedulers, in our proposed schemes, are structured in a tree form that we call a grid tree, as shown in Figure 1, where grid schedulers (GS) are placed into the tree according to their geographical locations. Users submit their jobs, in the form of requests to the grid via the grid system scheduler (GSS), which is the root node of the grid tree. A user's request describes the job in terms of the job minimum requirements (JMR) (e.g., operating system) in order to be matched to resources and includes any other constraints imposed by the user (e.g., completion deadline). A leaf grid scheduler (LGS), which is a node on top of local scheduler(s), connects directly with the user's workstation, brings the physical job to the grid (once a job is about to be mapped to the selected resources), and serves as middleware between the user's workstation and the allocated resources. Note that LGSs may be combined with local schedulers in one unit.

Theoretically, a scheduler, in our proposed systems, can break grid jobs into several subjobs to be executed in parallel and on different children's partitions. However, the art of automatic transformation into parallelism of an arbitrary job is in its infancy stage (Berstis, 2002). In our case, we assume whole jobs are submitted to resources.

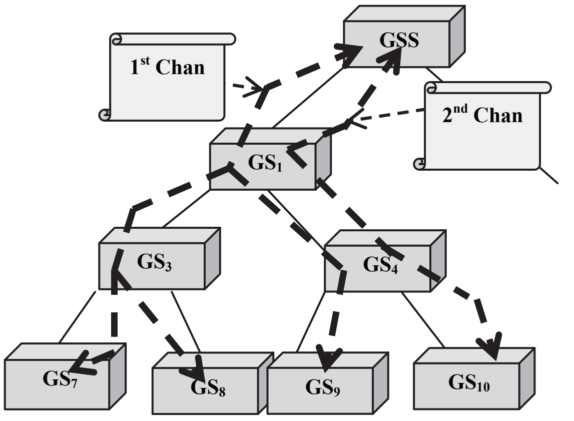
Resource Discovery Stage

In this stage, we propose the *self-discovery* method. The purpose of this method is to produce a set of logical channels to be used as paths by jobs (in the next scheduling stage) to get to their physical resources. Logical channels serve as a map for jobs so that they know how to reach resources that can meet their computational requirements. The self-discovery method omits irrelevant dissimilarities between resources of different sites. The principle behind this method is that resources are equivalent to each other, if they match the same set of jobs. For example, one site advertises INTEL architectures and another site advertises AMD architectures. Now, suppose the grid has a set of jobs that can be executed on either INTEL or AMD platform. In this case, the grid system considers architecture INTEL as equivalent to architecture AMD for those jobs in the set, since they can be executed on either platform. However, suppose now another set of jobs only requires architecture AMD in order to execute. In this case, the grid system considers architecture INTEL as nonequivalent to architecture AMD for the later set of jobs, since those jobs can only be executed on the AMD platform.

LGSs collect and store all static information about resources, either directly from local schedulers or from a GIS. Thus, information about local resources is distributed across the grid, which leads to (1) increasing system scalability and (2) maintenance of up-to-date databases. LGSs also initiate the resource discovery stage at system start up or when no jobs in the grid tree match their advertised resources, by issuing the request for job matching (RFJM) message to their parents, which in turn forwards the RFJM messages to the grandparents, and so on, until the RFJM message is received by the GSS, enabling it to initiate resource discovery to all of its raw jobs (i.e., new jobs that have not previously been through resource discovery stage). However, if the GSS has no raw jobs, it will then backlog the RFJM message until receiving new jobs.

The GSS starts the resource discovery stage by: (1) broadcasting a special message to all of its children to destroy all channels in the system and (2) passing all raw jobs to all of its children as one block. The children, in turn, pass the raw jobs as one block to the grandchildren, and so on, until they reach the LGSs at the bottom of the grid tree. LGSs match raw job requirements to their local resources and insert all raw jobs that match resources in their bags. They will

Figure 1. A grid tree with two channels



then pass a copy of their bags (along with any previous matched jobs) to their parents. Note that (1) LGSs save all requests that they receive from their parents, whether they have matched or not, enabling LGSs to perform rematching, if needed, due to resources change (i.e., the GSS removes all stored requests from all bags once they are executed); (2) intermediate schedulers (GS) always pass one RFJM message to their parents on behalf of their children and suppress other RFJMs, preventing the GSS from initiating any unnecessary resource discovery; and (3) a logical channel is created for every unique job bag.

Once a scheduler receives a bag (from one of its children) that is similar to another bag of an existing channel, it will then: (1) create a new branch from its existing channel and bind it with the child's channel; (2) recalculate its channel's processing power based on the new created branch; (3) inform the child of its channel's port number; and (4) update the parent, if needed, with the new processing power of its channel. However, if the received bag is distinctive, the scheduler will then: (1) create a new channel with a new port number; (2) create a new branch from its new channel and bind it with the child's channel; (3) initialize the channel processing power, based on the newly created branch; (3) inform the child with the channel's port number; and (4) update the parent, if needed, with the new bag and the channel's port number.

Now consider, as an example, the grid tree shown in Figure 1. Suppose the GSS pass six raw jobs, J_1 through J_6, as one block to all of its children. Suppose further that J_4, J_5, and J_6 do not match resources at GS_{10}. In this case, the GSS ends up with two channels (via GS_1), where all jobs may use the first channel. On the other hand, J_1, J_2,and J_3 are the jobs that can only use the second channel.

Resource Selection Stage

The grid AHS scheduling method, in this stage, uses *self-scheduling* by exploring the parent-child relationship. When a nonroot GS wants some work to do, it initiates self-scheduling by sending a request for computation (RFC) message to its parent (via a channel), requesting computation from it. If the parent GS does not have computations that can be pushed to that child's channel at the time of receiving the RFC, it, in turn, generates its own RFC and sends it to its parent on the next level of the grid tree. This process is recursively followed until either the RFC reaches the GSS or a GS with computations is encountered along the path. Note that intermediate schedulers send one RFC message per channel to their parents but still mark all branches from which they have received RFCs. Upon receiving an RFC message from a channel's branch of a child, a scheduler uses a space-sharing policy to distribute computations among channels as follows:

$$B_{share} = \lceil B_{rate} \times N \rceil, \qquad (1)$$

where B_{share} is the branch's share of all jobs within the scheduler's subtree; B_{rate} is the branch's transfer rate; and N is the number of jobs within a scheduler's subtree. B_{rate} is calculated as follows:

$$B_{rate} = \frac{B_{pwr}}{\sum_{i=1}^{M} (C_{pwr})_i}, \qquad (2)$$

where B_{pwr} is the branch's processing power; C_{pwr} is the channel's processing power (i.e., total processing power for all of its branches); and M is the number of channels in a scheduler. Note that, in our case, the channel's processing power is the number of central processing units (CPUs) that reside under that channel; since we assume that our computational resources are parallel computers (see the Simulation Model and Results section). However, in reality, we expect processing power to consider more factors, such as RAMs, bandwidth, and so forth.

Now, once a scheduler determines the number of jobs that will be pushed onto a channel's branch, it builds a list of those jobs as one block and pushes them onto that branch. Suppose, as an example, that GS_1, in Figure 1, has nine jobs, upon receiving an RFC message from the first branch of the first channel (i.e., via GS_3). Suppose further that the three branches that connect GS_1 with its two children have equivalent processing power. In this case, GS_1 may then push up to three jobs onto that branch.

Schedulers perform the following steps to collect the jobs (in order to be pushed onto a channel's branch): (1) they invoke the butterfly algorithm; (2) they collect jobs from the unassigned (i.e., unpushed) jobs; and (3) they invoke the load-balance algorithm. Note that we expect schedulers, in practice, to collect more dynamic information related to performance (e.g., load) or economics (e.g., prices).

Butterfly Algorithm

The principle behind this scheme is to reschedule jobs to better resources (with respect to predefined metrics), when they become available. Note that this algorithm can be extended to any soft conditions imposed by a user, where soft conditions are the ones that the user is willing to live without until they become available (or if they ever become available). In our case, we consider the geographic closeness of resources with respect to work stations as our metric (we use IP addresses to determine location nearness). Interestingly, a job may keep jumping (like a butterfly) among children's partitions, until it settles on the closest resources. In this algorithm, after a scheduler receives an RFC message from a child, the scheduler will then (1) cancel any assigned jobs from other children (if the new child is closer to those jobs' work stations) and (2) push them into the new available child's partition.

Fallback Algorithm

The fallback algorithm is intended to reschedule jobs that become incomputable because of the grid losing its required resources on its scheduled partitions. When an LGS detects resource change, it performs rematching for all saved requests. Now, if an LGS ends up with the same job bag, this resource change is then irrelevant. However, if it produces a different bag, it will then pass it on to its parent.

Schedulers handle received bags in this stage as previously described in the resource discovery stage. Additionally, schedulers mainly have to carry out the following (of course, parents will also be updated): (1) remove any jobs that become incomputable; (2) recalculate modified channels processing power; and (3) delete any broken channels. For example in Figure 1, assume GS_{10} changes resources and produces a bag similar to GS_9's bag. In this case, GS_4 will then connect GS_{10} to the first channel and inform its parent (GS_1) of two things: (1) the first channel with new updated processing power and (2) the broken second channel. Now if GS_4 has jobs that become incomputable, it will then remove them and update GS_1. Note that GS_1 reschedules those returned jobs (if they still computable on its partition) with a priority (in our case, the lesser the sequence number, the higher the priority). For instance, GS_1 may swap some of those returned jobs with assigned jobs in GS_3's partition, in order to execute jobs in the same order of their arrival to the grid.

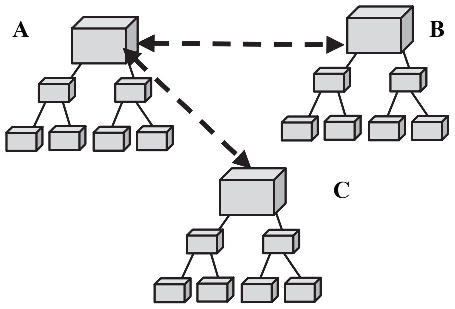
Load-Balance Algorithm

As stated earlier, schedulers determine the number of jobs (that will be pushed into a channel's branch) by considering all jobs within their subtrees. They will then collect those jobs by invoking the butterfly algorithm and from queued unassigned jobs. Schedulers will then balance the load among the channels by canceling already assigned jobs and then reschedule them on the channel's branch that just requested more work. Schedulers are always responsible for balancing all assigned jobs among their children's channels within their subtrees, since a parent may cause a child's subtree to get imbalanced (i.e., of course, parents do not know how assigned jobs are distributed within their children's subtrees). For example, assume that each of GS_7 and GS_8 in Figure 1 has four queued jobs (of course, GS_1 assumes that all eight jobs are still queued at GS_3). Suppose now that GS_1 decides to cancel four jobs to reschedule them on a GS_4's channel, in order to balance its subtree. Now, if GS_1 cancels the four jobs queued at GS_8, it imbalances the GS_3's subtree. In this case, GS_3 will balance its subtree upon receiving an RFC message from GS_8 (or will forward the RFC message to GS_1, if the RFC message is received from GS_7).

Hybrid System

The hierarchal system (one grid tree) has major drawbacks: (1) All requests are submitted to the GSS, which may become overwhelmed with too many requests; (2) it is difficult to bring in many organizations and have them agree on things, such as constructing the grid tree, controlling GSS policies (e.g., security) and dealing with

Figure 2. Hybrid system example



new joined organizations; and (3) it is difficult to convince companies to replace their peer-to-peer (P2P) based grid systems.

The hybrid system, which is several grid trees that act also as peers to each other, as shown in Figure 2, does not only overcome the above drawbacks but does provide organizations with more efficient ways to manage their own resources, such as stamping foreign requests with low priority, isolating their resources swiftly from the entire grid without pumping out their pending requests of using their resources, and so forth.

In the hybrid system, a GSS in a grid tree also acts as a peer GSS (PGSS) that forwards requests (after decrementing hop count) to its neighbors. Of course neighbors can be part of any other system types (e.g., P2P system). The P2P system can be viewed as a hybrid system, where each grid tree has only one scheduler; and the hierarchal system can be viewed as a hybrid system with one peer. In our case, we assume that (1) requests are always forwarded to neighbors (i.e., a request dies when hop count reaches 0), hence, grid trees also serve as backups to each other; and (2) foreign and home requests are queued in the same fashion.

SIMULATION MODEL AND RESULTS

This section presents the simulation model and samples of the preliminary results. Readers are encouraged to refer to Al-Zoubi (2006) for the complete set of results and a more in-depth discussion.

The grid simulation model is broken into three submodels—communication, node, and system models—each of which is described below.

Communication Model

The communication model, which is used by nodes (i.e., node model) to communicate with each other, consists of 2,400 nodes that span across four backbones. Each backbone (600 nodes) consists of four nets, where each net consists of 10 networks and each network consists of 15 nodes. Therefore, there are four backbones, four nets, 10 networks, and 15 nodes, which add up to 2,400 nodes in total in the model. The communication model uses the discrete event simulation (DEVS) CD++ simulator

(Al-Zoubi, 2006; Wainer, 2002) to simulate all of the communication aspects among all nodes.

The model presented numerous of challenges that we had to address to bring it closer as much as possible to the actual communication over the Internet, which is almost an impossible job to do, since the Internet is a very large unpredictable public network. We've assumed that 10% of the model's capacity accounts for the external Internet load and the routers processing time based on the studies in (Al-Zoubi, 2006; Odlyzko, 2003), which were based on actual statistics by the Internet Service Providers (ISP). We've also assumed 64 kilobytes TCP window size to control data flow (Al-Zoubi, 2006). Backbones in the model are connected with 1000 km, 9.6 Gb/s (e.g. OC-192 link) cables, Nets/sites are connected with 50 km, 155 Mb/s (e.g. OC3 link) cables (Al-Zoubi, 2006; Odlyzko, 2003), and nodes within a site are connected with 100 MBytes/s (Al-Zoubi, 2006) cables.

Node Model

A node, in the communication model, is simply a computer with an IP address. On the other hand, the node component contains the implementation of the proposed schemes for the grid systems in this article. A node can be configured to operate as a peer, local scheduler, intermediate GS, LGS, GSS, PGSS, or a work station. Note that the node's configuration determines the system model type.

System Model

The grid model can be configured to a (P2P, a hierarchal (one grid tree), or a hybrid (several grid trees) system model, as discussed below. The P2P systems are distributed systems and the only ones, to our knowledge, that are currently well thought-out by researchers (Al-Zoubi, 2006; Nabrzyski et al., 2004; Shan, Smith, Oliker, & Biswas, 2004) to replace the centralized systems. As a result, it is a reasonable choice to be compared against the proposed hierarchal systems in this article.

P2P System

Once a job request is received at a peer that meets its requirement, it contacts the work station to offer its service and amount of time for the work station response (100 ms, in our case). Work stations may accept peer service by responding to it or may refuse peer service by simply not responding to it. If a peer cannot accept a work station request, it decreases the hop count (1,000 in our case) in the message and forwards it to its neighbors. In our model, peers accept requests if they meet their deadline, which is three times the estimated execution time for that job. To improve P2P performance, we would suggest that: (1) if a workstation doesn't get a service offer within two minutes, it resubmits the job request to the grid; and (2) neighbors are manually configured to be geographically close. However, this may not be the case in reality.

Hierarchal System

The heirarchal system has one grid tree. The tree is constructed by connecting the GSS to four children, where each child holds one backbone. Each backbone's root has four children, where each child holds one net. Each net's root has two children, where each child holds five sites.

Hybrid System

The hybrid system has several grid trees acting as peers to each other. We use two hybrid systems in our simulation: four

Table 1. Computational resources

Server number	Number of Nndes	CPUs per node
1	184	16
2	305	4
3	144	8
4	1,024	4
5	64	2
6	512	4
7	128	2

grid trees system (Hybrid-4T) and 16 grid trees system (Hybrid-16T). In the Hybrid-4T system, the grid tree of the hierarchal system is broken into four grid trees, where each backbone has one grid tree. In the Hybrid-16T (16 grid trees), each net has one grid tree.

Grid Jobs

A work station submits a job to the grid via its grid entry (e.g., GSS) as a request that defines the JMR for that job. Jobs are assumed to be executed until completion of their predefined requirements (operating systems, in our case). Gaussian distribution is usually used to simulate the required time to run a job on a server (Al-Zoubi, 2006; Hotovy, 1996; Shan et al., 2004; Takefusa, 2001) with respect to the input job size and the server's processing power. Therefore, we assume job sizes are correlated to the amount of work performed by each job, where the input data size is expressed by Gaussian distribution with the mean $\mu = b * cpus * \text{wall time}$ in seconds, and where $b = 100$ (Shan, 2004). We also assume a job produces output data five times the original input job size.

Computational Resources

We assume all resources (i.e., servers) are parallel machines that consist of a number of interconnected nodes with a number of CPUs within a node as in Al-Zoubi (2006) and Shan et al. (2004). Table 1 shows the servers used in the simulation experiments, which are originally based on real machines (Al-Zoubi, 2006; Hotovy, 1996; Shan et al., 2004).

Those servers are duplicated in all the 160 sites in a range of two to six servers per site. The type of server and the operating system (Windows, UNIX, or LINUX) are picked at random. We assume 0.1 local loads (i.e., not related to the grid) on all computational resources at all times, as the typical case in most studies like Hotovy (1996). Note that the simulation model consists of 520 servers versus about 1,880 work stations throughout all experiments for a ratio of 1:3.6.

Workloads

Unfortunately it was difficult to find real traces for grid computing. However we were able to base our workloads on real traces for parallel machines and scientific applications (Al-Zoubi, 2006; Feitelson, 2005; Hotovy, 1996; LTTR, 2000; Shan et al., 2004). Jobs in the workloads, that are relevant to this article, use input average sizes of 1GB, 10GB, and 100GB over the following number of jobs: 520, 1,040, 1,560, 3,000, and 10,000 jobs. Refer to Al-Zoubi) (2006) for the complete set of workloads. We use Poisson distribution to generate input data sizes for submitted jobs to the grid, where the Poisson mean is set to the desired average input size. In this way, jobs are generated with different sizes but with the desired average input size, which is close to the typical case in reality.

Performance Metrics

We use three performance metrics to compare systems: total response time, average waiting time, and average response time.

The total response time (TRT) is the time from submitting first job request until the completion of all jobs in a workload. For example, suppose that the first request was submitted to the grid at 5 p.m. and the last job of a workload was completed at 10 p.m, the total response time will then be five hours. The purpose of this metric is to show the degree of parallelism in the grid, since we view the grid as a huge virtual parallel machine. The total response time (TRT) is calculated as follows:

$$TRT = (LJC - FRS)\,, \quad (3)$$

where LJC (last job completion time) is the time that of the output (i.e., at workstation) of the last completed job in the workload is received FRS (first request submission) is the time of transmission of the first request by a work station.

The waiting time (WT) for a job is the time from submitting the job's request to the grid until the start of the actual job transfer to the selected resources. For example, if a work station submits a request to the grid at 5 p.m. and gets a service offer from a resource at 6 p.m., the waiting time for that job is one hour. The purpose of this metric is to measure the scheduling time (i.e., the time it takes until a resource is allocated to that job). The average waiting time (AWT) is calculated as follows:

,

$$AWT = \frac{1}{N}\sum_{j=1}^{N}(SJTT - RT)_j \quad (4)$$

where N (job count) is the number of jobs in a workload. SJTT (start job transferring time) is the time when a workstation receives a service offer from a resource and starts transferring the physical job. RT (request time) is the time when a workstation submits that job request to the grid.

The execution time (ET) is the time from submitting the actual job to the grid until the job's output is received at the submitter's work station. For example, a work station receives a service offer from a resource at 5 p.m. Suppose now that the work station receives the output of the job at 6 p.m., then the execution time is one hour. Although all systems, in our model, function the same way when a request is mapped to a resource, we still need this metric to measure the location of where a job was executed. The average execution time (AET) is calculated as follows:

$$AET = \frac{1}{N}\sum_{j=1}^{N}(JCT - SJTT)_j \ , \qquad (5)$$

where JCT (job completion time) is the time when the job's output is received at the work station.

Simulation Experiments

We present, in this section, a sample of the experimental results to compare the performance over different scenarios. Refer to Al-Zoubi) (2006) for the complete set of results. Note that regardless of the configured system or experiment, the following assumptions still apply:

1. The computational power in the grid is maintained (i.e., 520 servers all the times);
2. a job is submitted by one work station and executed by one server. Note that a workstation is called active if it has a pending request in the grid. Otherwise, it is called inactive;
3. a work station that submits jobs according to a stochastic rate, only operates at that rate while it is inactive. For example, a work station submits jobs to the grid with the rate of 12 hours. Now, when that work station becomes inactive, it waits, according to that Poisson distribution, with a mean of 12 hours before it submits another job;
4. All results are obtained by averaging 20 different runs. Note that the difference between the worse and the best case runs is in the range of 5-15%. Perhaps, this is because of having too many nodes in the model.

First Experiment

In this experiment work stations submit jobs one after another until the entire workload is completed. This scenario is possible when an organization, for instance, executes a number of jobs one after another automatically as a set. The workload in this experiment is already distributed among sites by the submission approach. For example, if site A has three work stations and site B has six work stations. Most likely, site B will submit twice as many requests as site A.

The AWT and the TRT showed a substantial improvement against the P2P system, regardless of the number of used grid trees, workload, or scenario, as shown in Figures 3, 4, 6, and 9. Interestingly, the AWT starts declining when the hybrid system contains too many grid trees. Perhaps, this is because it gets closer and closer to the P2P system as a result of the increased number of trees in the system. The AET is almost the same for small jobs (1GB) but starts to differ when job size increases (100GB), as shown in Figures 5 and 7, which makes sense, since the model is built with high-performance links.

Second Experiment

In this experiment, work stations operate at different stochastic submission rates, where each work station selects, at random, one of the following rates: 10 minutes, 30 minutes, one hour, five hours, one day, or one week. Furthermore, a workload in this experiment is not already distributed among sites, as in the case of the first experiment. Furthermore, in this scenario, sites also have different probabilities when generating a job. For example, if site A has three workstations and site B has six, it is not necessarily true that site B is going to submit twice the number of requests that

Figure 3. A sample of AWT in experiment 1

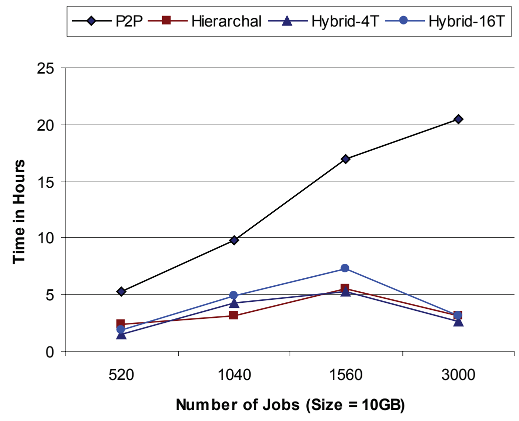


Figure 4. A sample of TRTime in experiment 1

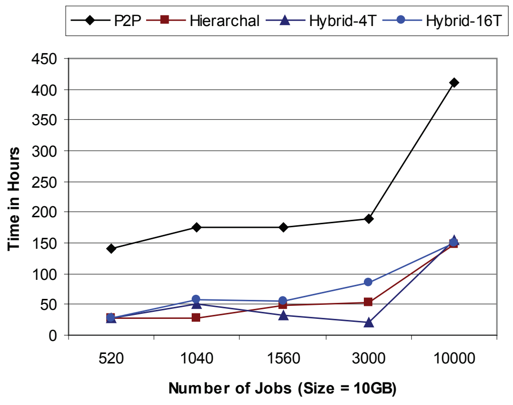
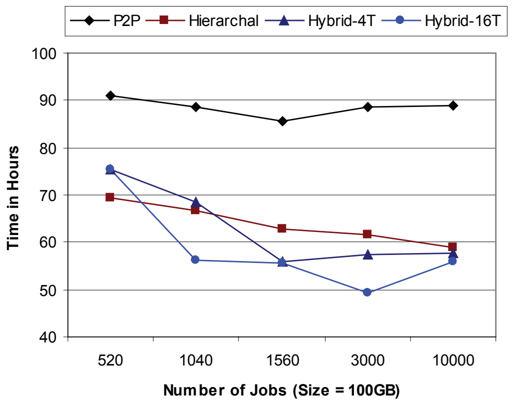


Figure 5. A sample of AET for large-sized jobs in experiment 1



will be submitted from site A. On the other hand, it is quite possible that all requests will be submitted from site A.

Observations of the first experiment also are supported by this experiment. Furthermore, both the butterfly and the load-balance algorithms showed a significant influence on the performance of the hierarchal system, as shown in Figure 8. In fact, the more jobs in a workload the worse it gets. If the subject algorithms are disabled, hence, the more jobs, and the more performed rescheduling.

Third Experiment

In this experiment work stations submit jobs with the same stochastic rate (e.g., a one hour rate for all work stations in the grid). A workload in this experiment is already distributed among the sites, as in the case of the first experiment. We studied the systems with three different rates: one hour, one day, and one week.

Observations of the previous experiments also are supported by this experiment. In addition, the AWT tends to decline with a big slope in the hierarchal systems, when jobs arrive into the grid with a larger mean rate. However, it decreases slightly in the P2P system, as shown in Figure 9.

Fourth Experiment

In this experiment, resources change according to one of the following stochastic changing-rates: one day, three days, one week, one month, three months, or six months. Note that the changing rate also is reselected at random, along with the advertised resources. For example, a server selects a six-month changing rate and reselects a three-month changing rate when it changes its advertised resources.

Now, when resource change is a possibility during job scheduling, the AWT turns out the same as when when resources are constant, as shown in Figure 10. This makes sense, since the fallback algorithm reschedules jobs, while resources are busy executing other jobs.

CONCLUSIONS

Many studies jump over the resource discovery stage into the second scheduling stage by assuming that all jobs can be executed anywhere in the grid or by simply assuming that resources will be discovered using the P2P approach. However, as we have shown, those stages have to be dealt with in a sequence because of their dependence on each other. The hierarchal approach has not only shown substantial improvement over the P2P system but also the ability to be combined with it in one hybrid system. Both the AWT and the TRT metrics showed a large improvement with the hierarchal approach in contrast to the P2P system, regardless of the number of grid trees, workload, or scenarios used. The AET metric also showed a significant improvement when not using the P2P system for large-sized jobs, but the numbers were almost the same for small-sized jobs. This makes sense, since the model is built with high-performance links. The three rescheduling algorithms showed a big contribution in the overall performance of the system. The fallback algorithm allowed some jobs to be executed despite resource change and maintained the same system performance when resources were constant. Both the butterfly and load-balance algorithms prevented the system from performing poorly when the number of jobs in workloads was increased.

Observably, the P2P approach puts the burden of discovering resources on the jobs. Peers "blindly" forward requests to

Figure 6. A sample of AWT in experiment 2

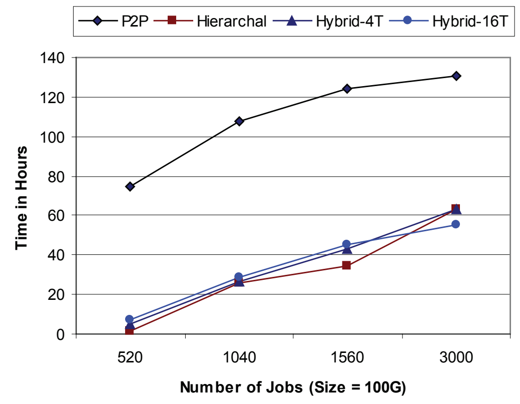


Figure 7. A sample of AET for small-sized jobs in experiment 2

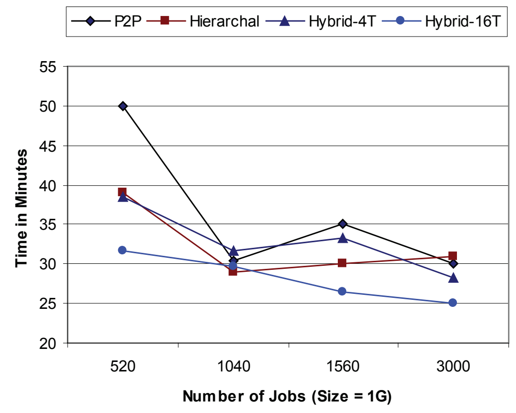


Figure 8. A sample of algorithms influence on the AWT in experiment 2

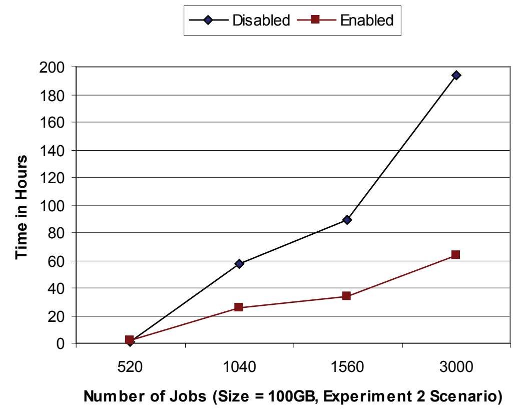


Figure 9. A sample of AWTime in experiment 3

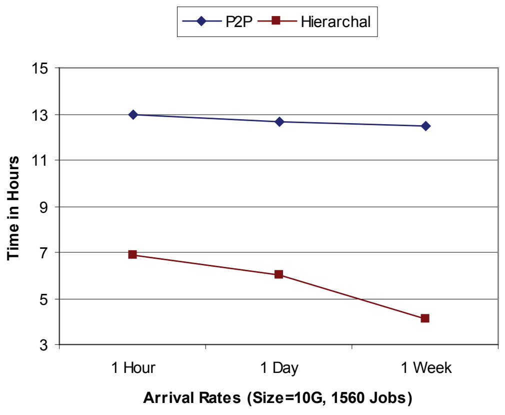
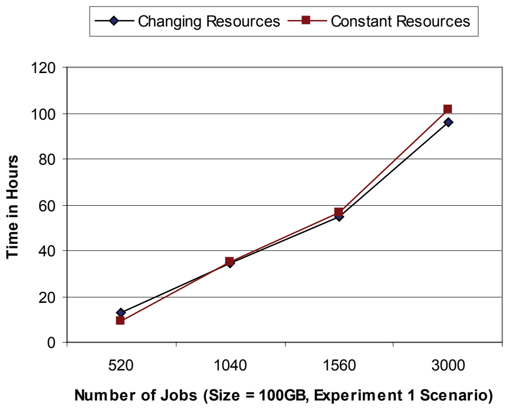


Figure 10. A sample of AWT in experiment 4



their neighbors with the hope that those jobs will find appropriate resources. However, as was shown in this article, the hierarchal approach gives schedulers more "say" in discovering resources for jobs and in distributing the jobs among resources. This is not a trivial issue if we want to gain the full benefits of the grid systems. Therefore, grid schedulers, in the future, need to break grid jobs into subjobs and execute them in parallel on multiple resources. Currently, we do not see how peers in the P2P-based grid system can carry out this task. However, in theory, any grid scheduler in a grid tree may break a job into subjobs and execute that job in parallel among its children's partitions. For a more in-depth discussion, see Al-Zoubi (2006).

REFERENCES

Al-Zoubi, K. (2006). *Hierarchical scheduling in grid systems*. Unpublished master's thesis, Carleton University, Ottawa, Canada.

Berstis, V. (2002). *Fundamentals of grid computing*. Retrieved from http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf

Dandamudi, S. (2003). *Hierarchical scheduling in parallel and cluster systems*. Kluwer Academic Publishers.

Feitelson, D. (2005). *Parallel workloads archive*. Available from http://www.cs.huji.ac.il/labs/parallel/workload/

Foster, I. (2001). The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. Retrieved from http://csdl2.computer.org/comp/proceedings/ccgrid/2001/1010/00/10100006.pdfFoster, I., & Kesselman, C. (2004). *The grid: Blueprint for a new computing infrastructure*. Morgan Kaufmann.

Hotovy, S. (1996). Analysis of the early workload on the Cornell theory. *ACM SIGMETRICS,* 272-273.

LTTR. (2000). *Long term technology review of the science & engineering base*. Available from http://www.rcuk.ac.uk/lttr/

Nabrzyski, J., Schopf, J., & Weglarz, J. (2004). *Grid resource management: State of the art and future trends*. Kluwer Academic Publishers.

Odlyzko, A. (2003). Internet traffic growth: Sources and implications. *Proceedings of the SPIE, 5247,* 1-15. Retrieved from http://www.dtc.umn.edu/~odlyzko/doc/itcom.internet.growth.pdf

Shan, H., Smith, W., Oliker, L., & Biswas, R. (2004). *Job scheduling in a heterogeneous grid environment.* Retrieved from http://www-library.lbl.gov/docs/LBNL/549/06/PDF/LBNL-54906.pdf

Takefusa, A. (2001). Bricks: A performance evaluation system for scheduling algorithms on the grids. *JWAITS,* .

Wainer, G. (2002). CD++: A toolkit to develop DEVS models. *Software: Practice and Experience, 32*(13), 1261-1306.

Khaldoon Al-Zoubi is a senior software system analyst and programmer. He has over 10 years in the telecommunications industry experience occupying a variety of software engineering and leadership positions in both the United States and Canada. He gained a wide range of software development experience in a number of areas including data link communications, simulation models, protocol stacks, client-server, real-time multi-tasking software, embedded software and Mobility. He holds a master' s degree in computer science in software engineering from Carleton University (Ottawa, Canada, 2006) and a Bachelor of Science in electrical and computer engineering from the University of Louisiana at Lafayette (Lafayette, Louisiana, USA, 1995). His school research is mostly focused in the area of job scheduling for parallel, cluster and grid systems.