Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences

Christopher Pinchak, Paul Lu, and Mark Goldenberg

Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada
{pinchak,paullu,goldenbe}@cs.ualberta.ca
http://www.cs.ualberta.ca/~paullu/Trellis/

Abstract. A practical problem faced by users of high-performance computers is: How can I automatically load balance my jobs across different batch queues, which are in different administrative domains, if there is no existing grid infrastructure? It is common to have user accounts for a number of individual high-performance systems (e.g., departmental, university, regional) that are administered by different groups. Without an administration-deployed grid infrastructure, one can still create a purely user-level aggregation of individual computing systems.

The Trellis Project is developing the techniques and tools to take advantage of a user-level *overlay metacomputer*. Because placeholder scheduling does not require superuser permissions to set up or configure, it is well-suited to overlay metacomputers. This paper contributes to the practical side of grid and metacomputing by empirically demonstrating that placeholder scheduling can work across different administrative domains, across different local schedulers (i.e., PBS and Sun Grid Engine), and across different programming models (i.e., Pthreads, MPI, and sequential). We also describe a new metaqueue system to manage jobs with explicit workflow dependencies.

Keywords: scheduling, metascheduler, metacomputing, computational grids, load balancing, placeholders, overlay metacomputers, metaqueue

1 Introduction

Metacomputing and grid computing are active research areas with the goal of developing the infrastructure and technology to create virtual computers from a collection of computers (for example, [7, 3, 8, 6, 12]). However, the constituent computers may be heterogeneous in their operating systems, local schedulers, and administrative control. The Trellis Project at the University of Alberta is addressing some of these issues through platform-independent systems to access computational resources [16, 18], remote data access [24], and scheduling [19, 9, 20]. The goals of the Trellis Project are not as comprehensive as other grid and metacomputing projects, but all of the projects share the goal of making it easier to take advantage of distributed computational and storage resources. In this paper, we extend our previous work [20] on the problems related to effectively scheduling computational tasks on computers that have different system administrators, especially in the absence of a single batch scheduler.

Table 1. Design Options for Grid and Metacomputer Scheduling

Design Option	Description	Main Advantages	Current Disadvantages
Metaqueue	Front-end queue that can redirect jobs to other queues. (E.g., routing queues in OpenPBS [17].)	Load balancing. Unified interface.	Requires common software systems, protocols, and administrative support.
Computational Grid	Common set of protocols and software infrastructure for metacomputing. (E.g., Globus Toolkit [8, 6] and Legion [12].)	Comprehensive set of features, including resource discovery and load balancing.	Relies on common grid infrastructure and cooperation of administrative domains. Generally speaking, unprivileged users cannot install or configure a grid.
User Scripts	Manual job placement and partitioning.	Simplicity.	Poor load balancing. Slow queue problem. Requires user intervention.
Placeholder Scheduling	User-level implementation of metaqueue. No special infrastructure or administrative support required.	Load balancing. Flexibility to create per-user and per-workload overlay metacomputers. Can be layered on top of existing (heterogeneous) queues, metaqueues, administrative domains, and grids.	Single job and advance reservations cannot span multiple queues or domains. No support for cross-domain resource discovery, etc.

1.1 Motivation: Overlay Metacomputers

Users often want to harness the cumulative power of an ad hoc collection of high-performance computers. Often, the problem is that each computer has a different

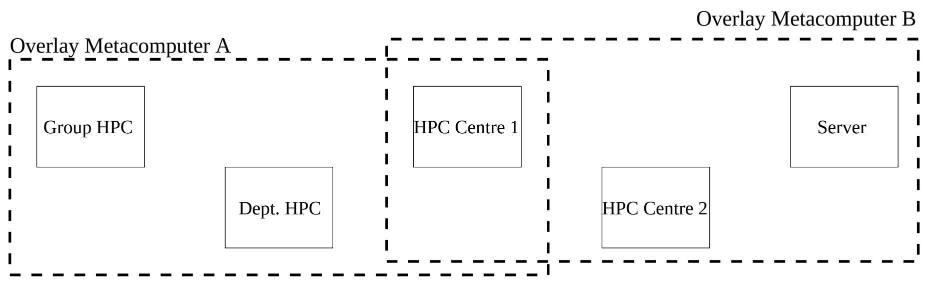

Fig. 1. Overlay Metacomputers

batch scheduler, independent queues, and different groups of system administrators. Two known solutions to this problem are: (1) implement a system-level metaqueue or (2) deploy a computational grid (Table 1).

First, if all of the individual computers are under a single group of system administrators, it would be possible (and preferable) to create a system-level metaqueue. For example, the OpenPBS implementation of the Portable Batch System (PBS) [17] supports *routing queues*. Similar capabilities exist in other workload management systems, such as Platform Computing's LSF [15]. Jobs are submitted to routing queues that decide which execution queue should receive the jobs. The advantage of a system-level and system-scheduled metaqueue is that more efficient scheduling decisions can be made. In effect, there is a single scheduler that knows about all jobs in all queues. Also, a system-level metaqueue would, presumably, be well-supported and conform to the security and sharing policies in force within the administrative domain. However, if the collection of computers with execution queues spans multiple administrative domains, it may be difficult and impractical to implement such a metaqueue. The disadvantage of a system-scheduled metaqueue is that the local system administrators may be required to relinquish some control over their queues. If the centres are located at different institutions, it can be difficult to obtain such administrative concessions.

Second, if the various system administrators can be persuaded to adopt a single grid infrastructure, such as Globus [8, 6], Legion [12], or Condor [3, 7], a metaqueue can be implemented as part of a computational grid. The advantage of computational grids is that they offer a comprehensive set of features, including resource discovery, load balancing, and a common platform. However, if the system administrators have not yet set up a grid, the user cannot take advantage of the grid features. Furthermore, what if a user has access to two systems that belong to two separate grids?

A practical problem that exists today is that many researchers have access to a variety of different computer systems that do not share a computational grid or a data grid (Figure 1). In fact, each of the individual systems may have a different local batch scheduler (e.g., OpenPBS, LSF, Sun Grid Engine [26]). The researcher merely has an account on each of the systems. For example, Researcher A has access to his group's system, a departmental system, and a system at a high-performance computing centre. Researcher B has access to her group's server and (perhaps) a couple of different high-performance computing centres, including one centre in common with Researcher A. It would be ideal if all of the systems could be part of one metacomputer or computational grid. But, the different systems may be controlled by different groups who may not run the same grid software. Yet, Researchers A and B would still like to be able to exploit the aggregate power of their systems.

Of course, the user can manually submit jobs to the different queues at different centres. In the case of user-scheduled jobs, the schedulers at each queue are unaware of the other jobs or queues. The user has complete control and responsibility for job placement and monitoring. Although this strategy is inconvenient, it is a common situation. The advantage is that different administrative groups do not have to agree on common policies; the user merely has to have an account on each machine. The disadvantage of user-scheduled jobs is that they are labour-intensive and inefficient when it comes to load balancing [20].

A better solution than manual interaction with the local schedulers is to create an *overlay metacomputer*, a user-level aggregate of individual computing systems (Figure 1). A practical and usable overlay metacomputer can be created by building upon existing networking and software infrastructure, such as Secure Shell (`ssh`) [1], Secure Copy (`scp`), and World Wide Web (WWW) protocols. Because the infrastructure is accessible at the user-level (or part of a well-supported, existing infrastructure) Researcher A can create a personal Overlay Metacomputer A. Similarly, Researcher B can create a personal Overlay Metacomputer B, which can overlap with Researcher A's metacomputer (or not).

1.2 Motivation: Placeholder Scheduling

Placeholder scheduling creates a user-level metaqueue that interacts with the local schedulers and queues of the overlay metacomputer. More details are provided in Section 2. Instead of a push model, in which jobs are moved from the metaqueue to the local queue, placeholder scheduling is based on a pull model in which jobs are dynamically bound to the local queues on demand. The individual local schedulers do not have to be aware of the user-level metaqueue (which preserves all of the local scheduler's policies) because only the placeholder *jobs* have to communicate with the user-level metaqueue; the *local scheduler* does not interact with the metaqueue.

Placeholder scheduling has three main advantages. First, the user-level metaqueue is built using only standard software or well-supported infrastructure. Software systems that require a lot of new daemons, applications, configuration, and administration are less likely to be adopted and supported by a wide community. Our system is layered on top of existing secure network infrastructure (i.e., Secure Shell) and existing batch scheduler systems (i.e., we use OpenPBS [17] and Sun Grid Engine [26]). Second, placeholder scheduling does not require superuser privileges or special administrative support. Different users can create private metaqueues that can load balance across different systems. Third, user-level metaqueues have similar load balancing benefits to system-level metaqueues, except that placeholder scheduling works across heterogeneous systems even if the different administrators do not have common scheduling infrastructure or policies. In the absence of a system-level metaqueue or a computational grid, which is still the common case, placeholder scheduling can still be used to load balance jobs across multiple queues.

1.3 Contributions

In our previous work [20], we described a prototype implementation of placeholder scheduling and a set of experiments. That was a proof-of-concept system and empirical evidence for the efficacy of placeholder scheduling. This paper extends our previous work and contributes to the practical aspects of computational grids and metacomputing by detailing a new implementation of placeholder scheduling that:

1. **Works across three different administrative domains, none of which are part of the same system-level grid or metacomputer.** We use systems located in our department, at the University of Alberta's high-performance computing centre, and at the University of Calgary.
2. **Works with different local batch scheduler systems.** Our previous experiments used only PBS. For the first time, we show how the Sun Grid Engine can interoperate with our user-level metaqueue as easily as PBS.
3. **Can use an SQL database**, instead of a flat file, to maintain the state of the user-level metaqueue. The original flat file approach is still supported and used when appropriate. The SQL-based option adds the benefits of sophisticated concurrency control and fault tolerance. We have also implemented **support for specifying and maintaining workflow dependencies between jobs**. Therefore, as with a dataflow model, all jobs of a larger computation can be submitted to the system, but jobs will only be executed when their predecessor jobs have been completed.
4. **Includes dynamic monitoring and throttling of placeholders.** We demonstrate a simple but effective system for controlling the number of placeholders in each local queue. When the local system is lightly loaded, more placeholders are created in order to maximize the throughput of the metaqueue. When the local system is heavily loaded, fewer placeholders are used because there is no benefit in having more placeholders.

2 Placeholders

2.1 The Concept

A placeholder can be defined as a unit of potential work. For an actual unit of work (i.e., a job), it is possible for any placeholder, within a group of placeholders, to actually complete the work. For example, in Figure 2, six placeholders

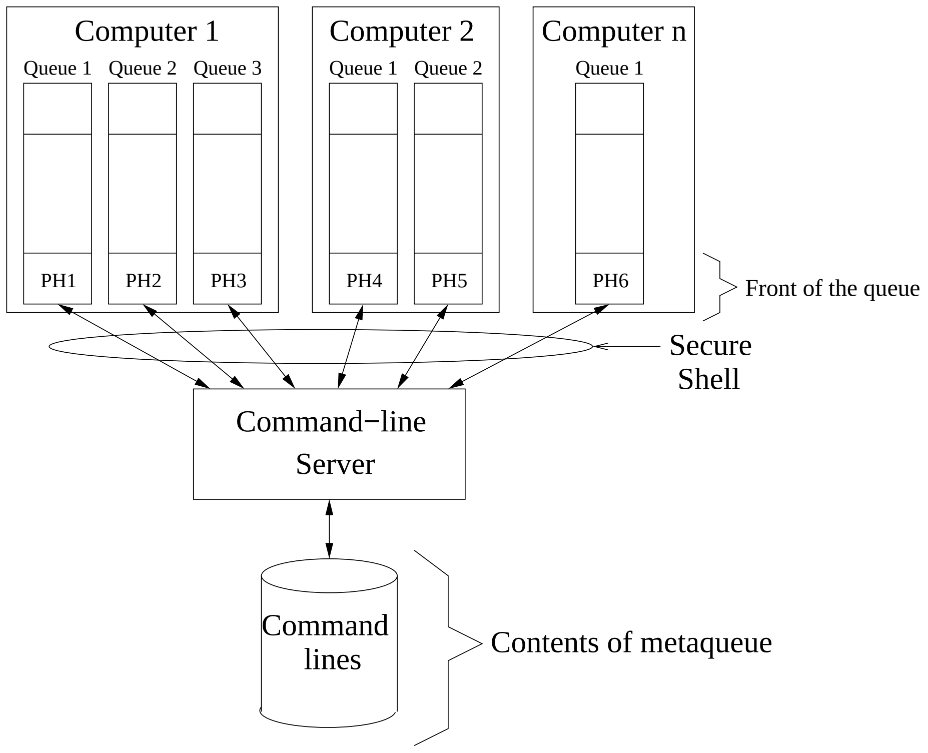

Fig. 2. Placeholder System Architecture

(i.e., PH1 to PH6) have been submitted to six different queues on three different computer systems. Any one of the placeholders is capable of executing the next job in the metaqueue. The run-time binding of placeholder to job occurs at placeholder *execution* time (not placeholder *submission* time) under the control of a command-line server (discussed in Section 2.2). We provide the implementation details in Section 3, but for now, one can think of a placeholder as a specially-crafted job submitted to the local batch scheduler. The placeholder job does *not* have any special privileges.

The first placeholder to request a new unit of work is given the next job in the metaqueue, which minimizes the mean response time for that job. The placeholder "pulls" the job onto the local computer system. Ignoring fault-tolerance, the same job is never given to more than one placeholder, and multiple placeholders can request individual jobs from a single metaqueue containing many jobs. If there are no jobs in the metaqueue when the placeholder begins execution, it can either exit the local queue or it can re-submit itself to the same queue. Informally, if there is no work to give to a placeholder when it reaches the front of the queue, the placeholder can go back to the end of the line without consuming a significant amount of machine resources. Other practical aspects of placeholder management are discussed in Section 6.

All placeholders that are submitted to any system are done so on behalf of the user (i.e., the jobs belong to the user's account identity). Therefore, all per-user resource accounting mechanisms remain in place. Some metacomputing systems execute jobs submitted to the metaqueue under a special account. We preserved submission from user accounts for three reasons: (1) job priority, (2) job accounting, and (3) security. Some sites base job priority on information about past jobs submitted by the user; other sites record this information for accounting (and possibly billing) purposes. Finally, security breaches of user accounts are significantly less dangerous than those of a superuser or privileged account.

2.2 Command-Line Server

The command-line server controls what executables and arguments should be executed by the placeholders. As an intermediary between the placeholders and the user-level metaqueue, it is possible for users to dynamically submit jobs to the command-line server and be assured that, at some point, a placeholder will execute the job.

We have augmented the command-line server with the ability to sequence jobs (and their respective command-line arguments) according to workflow dependencies. When jobs are submitted to the metaqueue, which is used by the command-line server, the user can optionally list job dependencies. Jobs cannot be assigned to placeholders (i.e., executed) until the predecessor jobs have been completed. Consequently, jobs may be executed in an order different from that in which they were submitted to the metaqueue, but the order of execution is always with respect to the required workflow.

3 Implementation

The basic architecture of our system is presented in Figure 2. We use the Secure Shell [1] for client-server communication across networks and either OpenPBS [17] or Sun Grid Engine [26] for the local batch schedulers. In our simple experimental system, placeholders contact the command-line server via Secure Shell. Placeholders use a special-purpose public-private key pair that allows it to authenticate and invoke the command-line server on the remote system.

All placeholders within the experimental system are submitted using the same user accounts. Currently, the placeholders and command-line server execute under normal user identities that do not have any special privileges. In fact, as discussed above, it is important that the placeholders are submitted via the user account to allow for proper prioritization and accounting at the local queue. And, should a malicious user acquire the private key of the placeholder, the damage would be limited because normal user accounts are non-privileged.

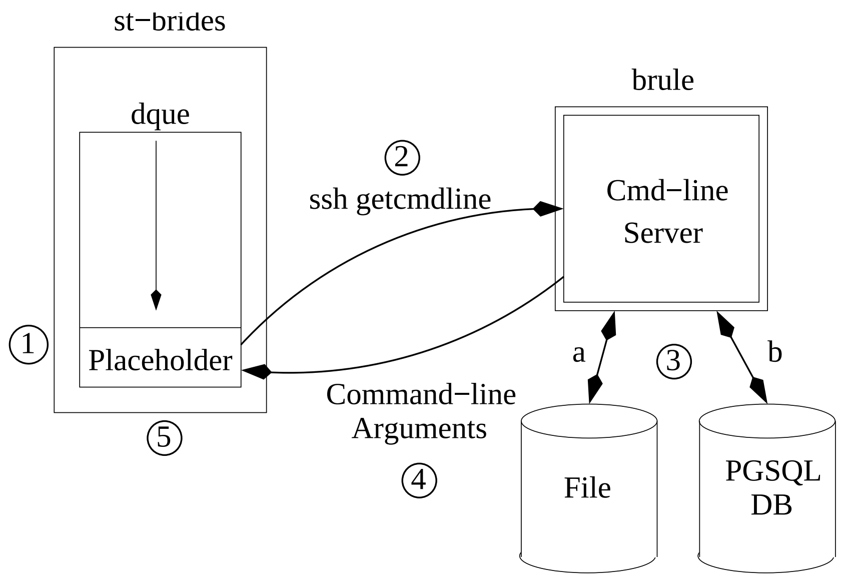

Fig. 3. Steps in Placeholder Execution

3.1 Example: Steps in Placeholder Execution

The flow of control for an example placeholder on the machine `st-brides` is shown in Figure 3. The actions this placeholder takes before executing are as follows:

1. The placeholder reaches the front of the batch scheduler queue `dque`.
2. The placeholder script contacts the command-line server on machine `brule` via Secure Shell. The name of the current machine (`st-brides`) is sent along as a parameter.
3. The command-line server retrieves the next command line. Command lines are stored in either (a) a flat file (as with the parallel sorting application described in Section 3.3), or in (b) a PostgreSQL [21] database (as with the checkers database application described in Section 3.4).
4. The results of the query are returned to the waiting placeholder. In the event that there are more command lines available, but none can be assigned because of dependencies, the placeholder is instructed to wait a short time and resubmit itself. If no more command lines are available, a message is sent notifying the placeholder to terminate without further execution.
5. The placeholder uses the returned command line to begin execution.

3.2 Dynamic Monitoring and Throttling of Placeholders

Because placeholders progress through the queue multiple times, it may be advantageous to consider the queue waiting time of the placeholder. Waiting time information may be utilized in order to decide how many placeholders to simultaneously maintain in a given queue. Low waiting times indicate that the queue is receiving "fast" service, and it may be a good idea to submit multiple placeholders to take advantage of the favourable conditions. For example, on a multiprocessor system, it may be possible to have different jobs execute concurrently on different processors; one job per placeholder. Conversely, high waiting times indicate that the queue is "slow" for the placeholder parameters and little will be gained by increasing the number of placeholders in the queue. Also, one does not want to have too many placeholders in the queue if the queue is making slow progress, lest they interfere with other users. This ability to throttle the number of placeholders may further reduce the makespan of a set of jobs.

3.3 Parallel Sort

A sorting application was chosen because of ease of implementation and because it may be implemented in a variety of different ways. Sorting may be done sequentially using a well-known efficient sorting algorithm (in our case, QuickSort), and in parallel (we used, Parallel Sorting by Regular Sampling (PSRS) [14]). Additionally, PSRS may be implemented in both a shared and distributed memory environment, allowing it to perform a sort on a variety of parallel computers. The variety of platforms on which a sort can be performed allows us to experiment with heterogeneous placeholder scheduling, with respect to the programming model.

A generic PBS placeholder is shown in Figure 4. The placeholder includes the ability to dynamically increase and decrease the number of placeholders in the queue. As illustrated, a placeholder is similar to a regular PBS job script. The lines beginning with `#PBS` (lines 4-11, Figure 4) are directives interpreted by PBS at submission time. The command line is retrieved from the command-line server (in our case, using the program `getcmdline`) and stored into the `OPTIONS` shell variable (line 18, Figure 4). This variable is later evaluated *at placeholder execution time* with the command(s) that will be executed (line 50, Figure 4). The late binding of placeholder to executable name and command-line arguments is key to the flexibility of placeholder scheduling.

The placeholder then evaluates the amount of time it has been queueing for (line 32, Figure 4), and consults a local script to determine what action to take (line 38, Figure 4). It may increase the placeholders in the queue by one (lines 44-47, Figure 4), maintain the current number of placeholders in the queue by resubmitting itself after finishing the current command line (line 60, Figure 4), or decrease the number of placeholders in the queue by not resubmitting itself after completing the current command line (lines 53-55, Figure 4).

Likewise, the basic command-line server is simple. Command lines themselves are stored in flat files, and the command-line server is implemented as a C program that accesses these files as a consumer process. Each invocation of the command-line server removes exactly one line from the flat file, which contains the arguments for one job. Each request to the command-line server invokes a

```
#!/bin/sh
## Generic placeholder PBS script

#PBS -S /bin/sh
#PBS -q queue
#PBS -l ncpus=4
#PBS -N Placeholder
#PBS -l walltime=02:00:00
#PBS -m ae
#PBS -M pinchak@cs.ualberta.ca
#PBS -j oe

## Environment variables:
##   CLS_MACHINE - points to the command-line server's host.
##   CLS_DIR - remote directory in which the command-line server is located.
##   ID_STR - information to pass to the command-line server.
## Note the back-single-quote, which executes the quoted command.
OPTIONS=`ssh $CLS_MACHINE "$CLS_DIR/getcmdline $ID_STR"`

if [ $? -ne 0 ]; then
    /bin/rm -f $HOME/MQ/$PBS_JOBID
    exit 111
fi
if [ -z $OPTIONS ]; then
    /bin/rm -f $HOME/MQ/$PBS_JOBID
    exit 222
fi

STARTTIME=`cat $HOME/MQ/$PBS_JOBID`
NOWTIME=`$HOME/bin/mytime`
if [ -n "$STARTTIME" ] ; then
    let DIFF=NOWTIME-STARTTIME
else
    DIFF=-1
fi

## Decide if we should increase, decrease, or maintain placeholders in the queue
WHATTODO=`$HOME/decide $DIFF`

if [ $WHATTODO = 'reduce' ] ; then
    /bin/rm -f $HOME/MQ/$PBS_JOBID
fi

if [ $WHATTODO = 'increase' ]; then
    NEWJOBID=`/usr/bin/qsub $HOME/psrs/aurora-pj.pbs`
    $HOME/bin/mytime > $HOME/MQ/$NEWJOBID
fi

## Execute the command from the command-line server
$OPTIONS

## leave if 'reduce'
if [ $WHATTODO = 'reduce' ] ; then
    exit 0
fi

/bin/rm -f $HOME/MQ/$PBS_JOBID

## Recreate ourselves if 'maintain' or 'increase'
NEWJOBID=`/usr/bin/qsub $HOME/psrs/aurora-pj.pbs`

$HOME/bin/mytime > $HOME/MQ/$NEWJOBID
```

Fig. 4. Generic PBS Placeholder

```
#!/bin/sh

## Checkers DB Placeholder PBS script

#PBS -S /bin/sh
#PBS -N CheckersPH
#PBS -q dque
#PBS -l ncpus=1
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -M pinchak@cs.ualberta.ca
#PBS -m n

OPTIONS=`ssh $CLS_MACHINE $CLS_DIR/next_job.py $ID_STR`

RETURNVAL="$?"

if [ "$RETURNVAL" -eq 2 ]; then
 exit 111
fi
if [ "$RETURNVAL" -eq 1 ]; then
    sleep 5
    qsub checkers_script.pbs
    exit
fi
if [ -z "$OPTIONS" ]; then
    exit 222
fi

cd $CHECKERS_DIR
$OPTIONS

ssh $CLS_MACHINE $CLS_DIR/done_job.py $ID_STR

qsub checkers_script.pbs
```

Fig. 5. PBS Placeholder for Computing Checkers Databases

new process, and mutual exclusion is implemented using the `flock()` system call.

3.4 Checkers Database

The checkers database program is an ongoing research project that aims to compute endgame databases for the game of checkers [10]. For this paper, we are only concerned with the application-specific workflow properties of the computation. The placeholders for this application are simpler than in the previous example as they are not capable of regulating the number of jobs in the queue (see Figures 5 and 6; note the similarities between the placeholder scripts for PBS and SGE). For our experiment, the local computer systems are uniprocessors and they are dedicated to the computation. Therefore, there is little advantage in having more than one placeholder per queue.

The databases are computed using retrograde analysis [10]. To create parallelism and reduce memory requirements, the databases are logically divided into individual jobs, called slices. We denote a slice using four numbers. These

```
#!/bin/sh

## Checkers DB Placeholder SGE script

#$ -S /bin/sh
#$ -N CheckersPH
#$ -j y
#$ -M pinchak@cs.ualberta.ca
#$ -m n

OPTIONS=`ssh $CLS_MACHINE $CLS_DIR/next_job.py $ID_STR`

RETURNVAL="$?"

if [ "$RETURNVAL" -eq 2 ]; then
 exit 111
fi
if [ "$RETURNVAL" -eq 1 ]; then
    sleep 5
    qsub checkers_script.sge
    exit
fi
if [ -z "$OPTIONS" ]; then
    exit 222
fi

cd $CHECKERS_DIR
$OPTIONS

ssh $CLS_MACHINE $CLS_DIR/done_job.py $ID_STR

qsub checkers_script.sge
```

Fig. 6. Sun Grid Engine Placeholder for Computing Checkers Databases

numbers stand for the number of black kings, white kings, black checkers and white checkers. The slices are further subdivided into smaller slices based on the position of the most advanced checker of each side (see [10] for details). Because the results of one slice may be needed before another slice can be computed, there is an inherent workflow dependency.

Figure 7 shows the dependencies between slices of the databases for the case in which black has three pieces and white has two pieces on the board. For example, consider a position with 2 black kings, 2 white kings, 1 black checker and no white checkers. This position is in slice "2 2 1 0" of the databases. Now, if a black checker advances to a king, then we have 3 black kings, 2 white kings and no checkers. The new position is in slice "3 2 0 0". Thus, positions in slice "2 2 1 0" can play into positions in slice "3 2 0 0". This is reflected by an edge at the top of Figure 7. Therefore, slice "3 2 0 0" has to be computed before slice "2 2 1 0".

In general, slices at the same level of the lattice in Figure 7 can be computed in parallel; slices at different levels of the lattice have to be computed in the proper order (i.e., from top to bottom).

Information about the dependencies between board configurations is conveniently stored in a Makefile. This Makefile is automatically produced by a C

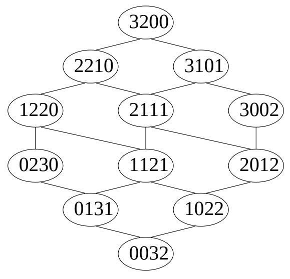

Fig. 7. Dependencies Between Slices of the Checkers Endgame Databases

```
CREATE TABLE Targets (
        tar_id          int PRIMARY KEY,
        tar_name        varchar(64) UNIQUE
);
CREATE TABLE Jobs (
        tar_id          int REFERENCES Targets, -- target ID
        j_num           int,                    -- number within target
        comm_line       varchar(800),           -- command line
        PRIMARY KEY (tar_id, j_num)
);
CREATE TABLE  Before (
        pre_id          int REFERENCES Targets ON DELETE CASCADE, -- prerequisite
        dep_id          int REFERENCES Targets,                   -- dependent target
        PRIMARY KEY (pre_id, dep_id)
);
CREATE TABLE Running (
        tar_id          int REFERENCES Targets,                   -- target ID
        j_num           int,                                      -- number within target
        machine         varchar(20),                              -- host name
        PRIMARY KEY (tar_id, j_num)
);
```

Fig. 8. Definition Script for Jobs Database

program. Commands in the Makefile are calls to a script (called `mqsub.py`) that inserts job descriptions and dependencies into a simple relational database (i.e. PostgreSQL [21]). The schema definition script is shown in Figure 8. An example of the submission of a job to the database is shown in Figure 9. We provide a name (or label) for the current target (or job) (following `-l`), the labels of the jobs on which the current job depends (following `-deps`), and the command line for computing the slice (following `-c`).

Tuples in the `Targets` table (Figure 8) correspond to targets in the Makefile. Commands within targets are assigned consecutive numbers. Thus a command is uniquely identified given its target ID and job number within the target (see table `Jobs`). Table `Before` summarizes information about dependencies between targets. Table `Running` contains the jobs that are currently being run; for each such job, the host name of the machine on which the job is being run is stored.

```
./mqsub.py -deps "0022 0031" -l "3200" -c "Bin/run.it 3 2 0 0 0 0 >& Results/3200.00"
```

Fig. 9. Submission of the Job For Computing a Slice in "3 2 0 0"

Table 2. Experimental Platform for the Parallel Sorting Application

System	Description	Interconnect	Scheduler	Method
A (`aurora`)	SGI Origin 2000, 46 × 195 MHz R10000, 12 GB RAM, Irix 6.5.14f	Shared Memory NUMA	PBS	Parallel Shared Memory
B (`lacrete`)	Single Pentium II, 400 MHz, 128 MB RAM, Linux 2.2.16	None	Sun Grid Engine	Sequential
C (`maci-cluster`)	Alpha Cluster, mixture of Compaq XP1000, ES40, ES45, and PWS 500au, 206 processors in 122 nodes, each node has from 256 MB to 8 GB RAM, Tru64 UNIX V4.0F	Gigabit Ethernet	PBS	Parallel Distributed Memory (i.e., MPI)

The command-line server consults and modifies the database of jobs in order to return the command line of a job that can be executed without violating any dependencies. The command-line server is invoked twice for each job: once to get the command line for the job (Figure 5, line 14) and the other to let the server know that the job has been completed (Figure 5, line 33). Both times, the host name is passed as a parameter to the server.

The design of the jobs database simplifies the task of the command-line server. All prerequisites for a target are met if the target does not appear in the `dep_id` field of any tuple in the `Before` table. Also, when the last job of a target is returned, the target is deleted from the `Targets` table, which results in cascading deletion of the corresponding tuples in the database.

4 Experiments

4.1 Parallel Sort

The goals of the parallel sorting experiment are to show the performance of placeholders in four orthogonal dimensions of heterogeneity: (1) parallel vs. sequential computer; (2) machine architecture; (3) distributed vs. shared memory; and (4) local scheduling system. A summary of the systems with respect to these dimensions is shown in Table 2.

We performed an on-line experiment with three different computers, in three different administrative domains, and with three different local schedulers. These are not simulated results. System A sorted four million integer keys using four processors, System B sorted four million integer keys sequentially, and System C sorted four million integer keys using eight processors. During our experiment, there were other users on two of the systems (i.e., System A and C). Although the specific quantitative results are not repeatable, the qualitative results are representative. Also, note that System A is administered by the high-performance computing centre of the University of Alberta. System B is in our department and effectively under our administrative control. System C is administered by the of the University of Calgary.

The primary goal of placeholder scheduling is to maximize throughput across a number of machines. The throughput, as evidenced by the rate of execution, is shown in Figure 10. The cumulative number of work units performed by each system is shown, and the rate of execution is determined by the slopes of the lines. System A exhibits a good initial execution rate, but then suddenly stops executing placeholders. System B, the dedicated sequential machine, exhibits a steady rate of execution. System C is somewhere in between, exhibiting a more or less constant rate of execution, although this rate is below that of the others. The bottom-most (bar) graph in Figure 10 shows the number of work units completed per 5000 second time period.

An interesting point illustrated in Figure 10 is the abrupt halt of execution of System A. By examining the PBS logs, we believe that our placeholders used up our user account's quota of CPU time on the system. As a result, System A becomes unable to execute additional work after roughly 7000 seconds, and this can be perceived as a failure of System A. However, because of the placeholders, the other two systems (B and C) are able to compensate for the loss of System A. After 7000 seconds, only Systems B and C complete work units and are responsible for finishing off the remainder of the workload. Should the loss have occurred without a scheduling system such as placeholder scheduling, users would likely have to discover and correct for this loss on their own.

Figures 11 and 12 show the queue lengths and placeholders per queue, respectively. As Figure 11 shows, System A is significantly more loaded than System C. However, System A is also more powerful than System C, and therefore execution rates are higher. System A is also able to sustain more placeholders in the queue for the first 7000 seconds, and both queues exhibit increases and decreases in placeholder counts due to changing queue conditions (Figure 12). It must be emphasized that these results are obtained from computers working on other applications in addition to our own. No attempt has been made to control the queues on Systems A or C.

4.2 Checkers Database

The purpose of the checkers database experiment is twofold. First, the checkers database application is a non-trivial application. Second, the computation of one slice is dependent upon the completed computation of other slices. Therefore,

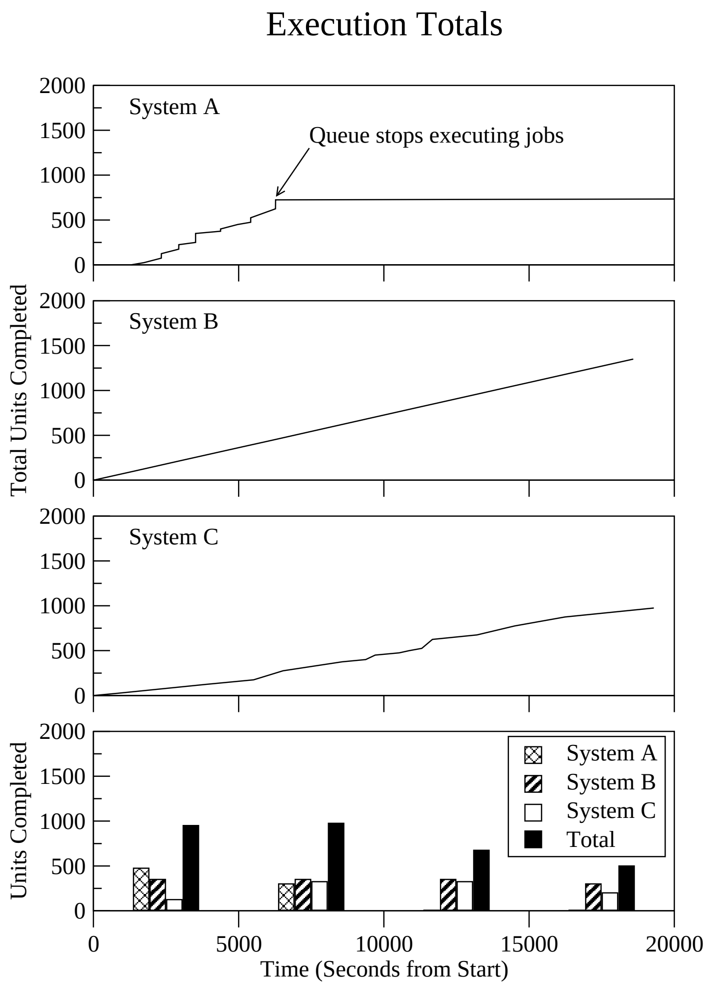

Fig. 10. Throughput for the Parallel Sorting Application

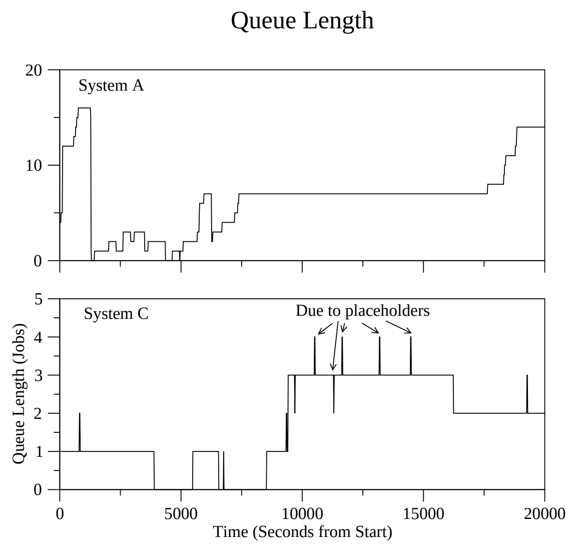

Fig. 11. Queue Lengths of the Parallel Machines

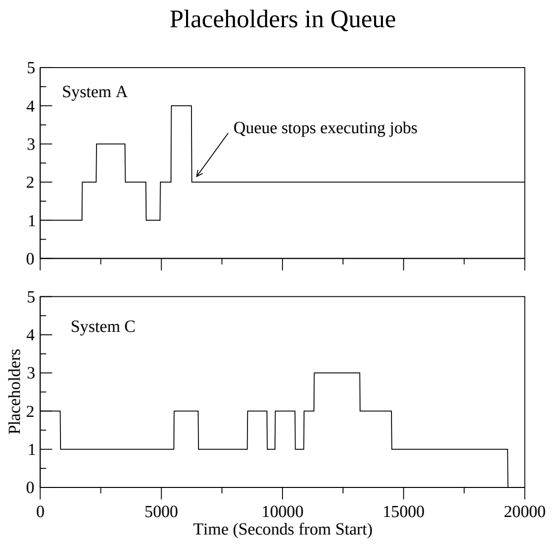

Fig. 12. Number of Placeholders in Parallel Machine Queues

Table 3. Experimental Platform for the Checkers Database Application

System	Description	Scheduler
D (`samson-pk`)	Single AMD Athlon XP 1800+, 256 MB RAM, Linux 2.4.9	Sun Grid Engine
E (`st-brides`)	Single AMD Athlon XP 1900+, 256 MB RAM, Linux 2.4.9	PBS

some form of workflow management must be present to coordinate the computation of board configurations. As was described above, a new command-line server was implemented to coordinate the computation.

Two different computers were used for computing the checkers databases (see Table 3). Figure 13 shows the throughput of the two computers in terms of the number of board configurations each computed. Because of the dependencies between some board configurations (see Figure 7), some board configurations must be computed sequentially. In our case, System E computes more of these sequential configurations than does System D. This is verified by the load averages shown in Figure 14. Overall, System E has a higher load, which indicates that it is performing more work.

Unlike the parallel sorting experiment, there are dependencies between jobs in the checkers database application. Furthermore, the number of jobs that can be computed concurrently varies from one (at the very top and bottom of the lattice) to a significant number (at the middle of the lattice) (Figure 7). Therefore, there are bottleneck jobs and the two computers are not fully-utilized during those bottleneck phases (Figure 14). However, when there are concurrent jobs, our placeholder scheduling system and the workflow-based command-line server are able to exploit it.

5 Related Work

The basic ideas behind placeholder scheduling are quite simple, but there are some important differences between placeholder scheduling and other systems.

In concept, placeholders are very similar to the GlideIn mechanism of Condor-G [7]. GlideIns are daemon processes submitted to the local scheduler of a remote execution platform using the GRAM remote job submission protocol of Globus [4]. As with placeholders, the local scheduler controls when the placeholder or daemon begins its execution and GlideIns also support the late binding of jobs to a computational resource.

In terms of implementation, placeholders and GlideIns have some significant differences. For example, placeholders do not require any additional software infrastructure beyond a batch scheduler and the Secure Shell, whereas Condor-G and GlideIns (as currently implemented) require the Globus infrastructure. Also, placeholders have a simple dynamic monitoring and throttling capability that is compatible with local schedulers. As previously discussed, one of the goals

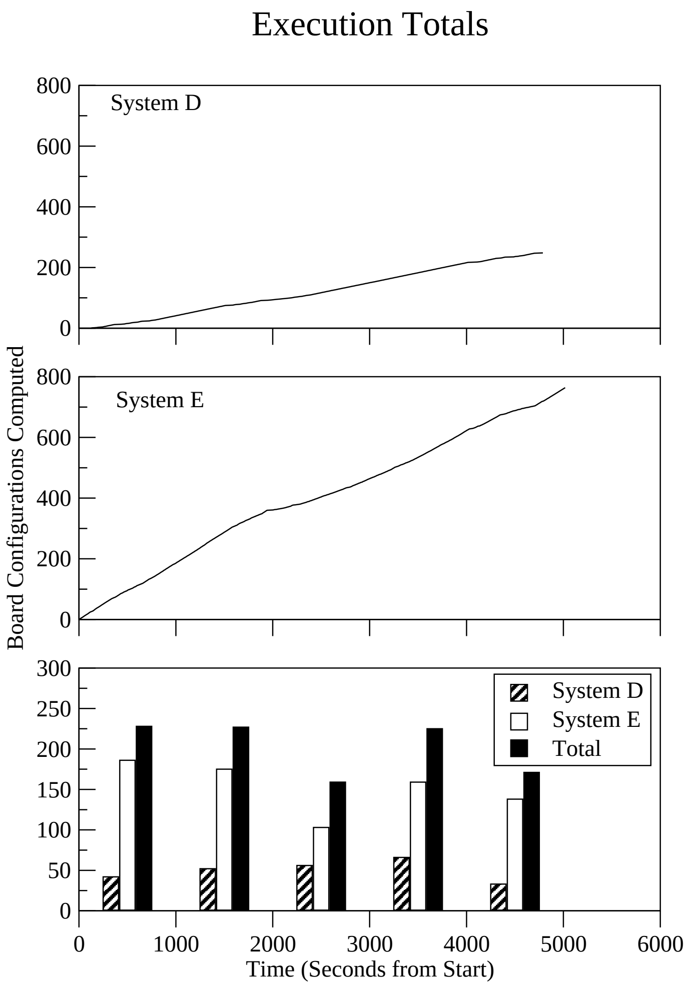

Fig. 13. Throughput for the Checkers Database Application

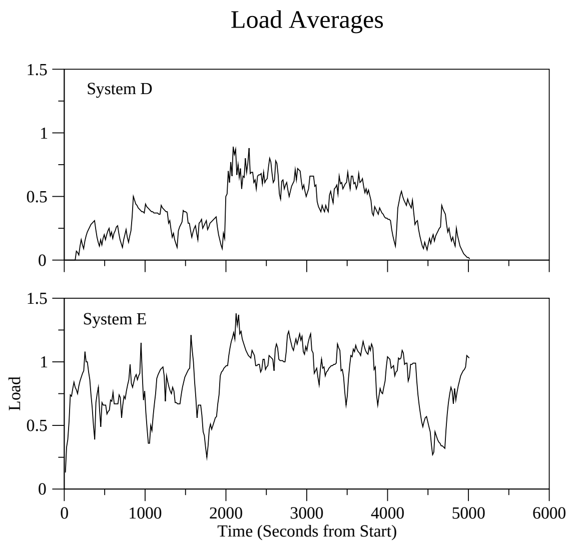

Fig. 14. Load Averages for the Checkers Database Application

of user-level overlay metacomputers is to only build upon existing networking and software infrastructure.

Of course, placeholders can be retargeted for Globus-based computational grids by using GridSSH [8] as a plug-in replacement for the standard Secure Shell. And, GlideIns can, in theory, be reimplemented for non-Globus-based grids.

More generally, we suspect that, prior to the availability of full-featured and open-source batch schedulers such as OpenPBS, most users wrote custom scripts to distribute their work (for example, [10]), without generalizing the system in the manner of this paper. We feel that our contribution is in demonstrating how placeholder scheduling can be implemented in a contemporary context and how it relates to metacomputing and computational grids. More tangentially, large-scale distributed computation projects such as SETI@home [23] use software clients that are, in essence, single-purpose placeholders that pull work on-demand from a server.

Placeholder scheduling shares many similarities with self-scheduling tasks *within* a parallel application and the well-known master-worker paradigm [22], in which placeholders are analogous to worker processes. Of course, our presentation of placeholder scheduling is in the context of job scheduling and not task scheduling. Nonetheless, the basic strategies are identical.

Of course, there is a large body of research in the area of job scheduling and queuing theory (for example, [5, 11, 13]). This paper has taken a more systemsoriented approach to scheduling. Our scheduling discipline at the metaqueue (i.e., command-line server) is currently simple: first-come-first-served. In the future, we hope to investigate more sophisticated scheduling algorithms that understand the dependencies between jobs and try to compute a minimal schedule.

6 Discussion and Future Work

In this section, we discuss some other important, practical aspects of placeholder scheduling. Many of the following issues are to be addressed as part of future work.

1. **Advanced Placeholder Monitoring.** We have implemented a simple form of placeholder monitoring and throttling. However, there are some other forms of placeholder monitoring that are also important and will be addressed in future work.
   Placeholders should be removed from local batch queues if the command-line server has no more jobs or too few jobs. We do not want a placeholder to make it to the front of the queue, allocate resources (which may involve draining a parallel computer of all the sequential jobs so that a parallel job can run), and then exit immediately when the command-line server has no work for it. A similarly undesirable situation occurs when there are fewer jobs in the metaqueue than there are placeholders.
   In both situations, placeholders should be automatically removed from the queues in order to minimize the negative impact that they might have on other users. If, later on, more work is added to the command-line server, placeholders can be re-started.
2. **Fault Tolerance.** Placeholders, by their nature, contain some amount of fault tolerance. Because placeholders are usually present in more than one queue, some queue failures (e.g., a machine shutdown or network break) can occur and the jobs will still be executed by placeholders in the remaining queues. However, a more systematic approach to detecting and handling faults is required to improve the practicality of placeholder scheduling.
   As part of advanced placeholder monitoring (discussed above), future placeholder scheduling systems have to monitor and re-start placeholders that disappear due to system faults. Also, the system should be able to allocate the same job to two different placeholders if a fault is suspected and, if both placeholders end up completing the job, deal with potential conflicts due to job side effects.
3. **Resource Matching.** Modern batch scheduler systems provide the ability to specify constraints on the placement of jobs due to specific resource requirements. For example, some jobs require a minimum amount of physical memory or disk space. Currently, our implementation of placeholder scheduling does not provide this capability, but it is an important feature for the future.
4. **Data Movement.** Another practical problem faced by users of metacomputers and computational grids is: If my computation can move from one

system to another, how can I ensure that my data will still be available to my computation?

Depending on the level of software, technical, and administrative support available, a data grid (for example, [2, 25, 27]) or a distributed file system (e.g., AFS, NFS) would be reasonable solutions. However, as with system-level metaqueues, it is not always possible (or practical) to have a diverse group of systems administrators agree to adopt a common infrastructure to support remote data access. Yet, having transparent access to any remote data is an important, practical capability.

Data movement is something that the Trellis Project has started to address. We have developed the Trellis File System (Trellis FS) to allow programs to access data files on any file system and on any host on a network that can be named by a Secure Copy Locator (SCL) or a Uniform Resource Locator (URL) [24]. Without requiring any new protocols or infrastructure, Trellis can be used on practically any POSIX-based system on the Internet. Read access, write access, sparse access, local caching of data, prefetching, and authentication are supported.

7 Concluding Remarks

The basic ideas behind placeholders and placeholder scheduling are fairly straightforward: centralize the jobs of the workload into a metaqueue (i.e., the command-line server), use placeholders to pull the job to the next available queue (instead of pushing jobs), and use late binding to give the system maximum flexibility in job placement and load balancing. Our contribution is in showing how such a system can be built using only widely-deployed and contemporary infrastructure, such as Secure Shell, PBS, and SGE. As such, placeholder scheduling can be used in situations in which metaqueues and grids have not yet been implemented by the administrators.

As an extension of our original work with placeholder scheduling, we have now empirically demonstrated that placeholder scheduling can (1) load balance a workload across heterogeneous administrative domains (Table 2), (2) work with different local schedulers (Table 2), (3) implement workflow dependencies between jobs (Section 3.4, Section 4.2), and (4) automatically monitor the load on a particular system in order to dynamically throttle the number of placeholders in the queue (Section 3.2).

Given the growing interest in metacomputers and computational grids, the problems of distributed scheduling will become more important. Placeholder scheduling is a pragmatic technique to dynamically schedule, place, and load balance a workload among multiple, independent batch queues in an overlay metacomputer. Local system administrators maintain complete control of their individual systems, but placeholder scheduling provides the same user benefits as a centralized meta-scheduler.

Acknowledgments

Thank you to C3.ca, the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Canada Foundation for Innovation (CFI) for their research support. Thank you to Lesley Schimanski, the anonymous referees, and the attendees of the 8th Workshop on Job Scheduling Strategies for Parallel Processing (Edinburgh, Scotland, July 24, 2002) for their valuable comments.

References

1. D. J. Barrett and R. E. Silverman. *SSH, the Secure Shell: The Definitive Guide.* O'Reilly and Associates, Sebastopol, CA, 2001.
2. J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999.
3. Condor. http://www.cs.wisc.edu/condor/.
4. K. Czajkowski, I. Foster, N. Karonis, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 62–82. Springer-Verlag, 1998.
5. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer-Verlag, 1997.
6. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.
7. J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, San Francisco, California, U.S.A, August 7–9 2001.
8. Globus. http://www.globus.org/.
9. M. Goldenberg. A System For Structured DAG Scheduling. Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, in preparation.
10. R. Lake, J. Schaeffer, and P. Lu. Solving Large Retrograde-Analysis Problems Using a Network of Workstations. In *Proceedings of Advances in Computer Chess 7*, pages 135–162, Maastricht, Netherlands, 1994. University of Limburg.
11. E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance. Computer Systems Analysis Using Queueing Network Models.* Prentice Hall, Inc., 1984.
12. Legion. http://www.cs.virginia.edu/~legion/.
13. M. R. Leuze, L. W. Dowdy, and K. H. Park. Multiprogramming a Distributed-Memory Multiprocessor. *Concurrency–Practice and Experience*, 1(1):19–34, September 1989.
14. X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19(10):1079–1103, October 1993. Available at http://www.cs.ualberta.ca/~paullu/.
15. Load Sharing Facility (LSF). http://www.platform.com/.

16. G. Ma and P. Lu. PBSWeb: A Web-based Interface to the Portable Batch System. In *Proceedings of the 12th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 24–30, Las Vegas, Nevada, U.S.A., November 6–9 2000. Available at http://www.cs.ualberta.ca/~paullu/.
17. OpenPBS: The Portable Batch System. http://www.openpbs.com/.
18. PBSWeb. http://www.cs.ualberta.ca/~paullu/PBSWeb/.
19. C. Pinchak. Placeholder Scheduling for Overlay Metacomputers. Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, in preparation.
20. C. Pinchak and P. Lu. Placeholders for Dynamic Scheduling in Overlay Metacomputers: Design and Implementation. *Journal of Parallel and Distributed Computing.* Under submission to special issue on Computational Grids.
21. PostgreSQL Database Management System. http://www.postgresql.org/.
22. L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation In Parallel Machines. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, South Carolina, U.S.A, July 21–24 1991. ACM Press.
23. SETI@home. http://setiathome.ssl.berkeley.edu/.
24. J. Siegel and P. Lu. User-Level Remote Data Access in Overlay Metacomputers. In *Proceedings of the 4th IEEE International Conference on Cluster Computing*, September 2002.
25. H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and Object Replication in Data Grids. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, San Francisco, California, U.S.A, August 7 – 9 2001.
26. Sun Grid Engine. http://www.sun.com/software/gridware/sge.html.
27. B.S. White, M. Walker, M. Humphrey, and A.S. Grimshaw. LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications. In *SC2001: High Performance Networking and Computing.*, Denver, CO, November 10–16 2001.