Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing

Niels Drost Rob V. van Nieuwpoort Henri Bal
Dept of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{ndrost, rob, bal}@cs.vu.nl

Abstract

With current grid middleware, it is difficult to deploy distributed supercomputing applications that run concurrently on multiple resources. As current grid middleware systems have problems with co-allocation (scheduling across multiple grid sites), fault-tolerance and are difficult to set-up and maintain, we consider an alternative: peer-to-peer (P2P) supercomputing. P2P supercomputing middleware systems overcome many limitations of current grid systems. However, the lack of central components make scheduling on P2P systems inherently difficult.

As a possible scheduling solution for P2P supercomputing middleware we introduce flood scheduling. *It is locality aware, decentralized, flexible and supports co-allocation. We introduce Zorilla, a prototype P2P supercomputing middleware system. Evaluation of Zorilla on over 600 processors at six sites of the Grid5000 system shows that flood scheduling, when used in a P2P network with suitable properties, is a good alternative to centralized algorithms.*

1. Introduction

With the ever increasing need for compute power in both science and industry, grid computing is seen as one of the best ways of having virtually unlimited processing power available to as many people as possible. Grid computing systems are generally made up of a collection of compute clusters, connected by a wide area network (WAN).

This paper examines the possibilities of performing *distributed supercomputing* on grid systems. Supercomputing applications use multiple computational resources simultaneously to perform one single computation. In the case of distributed supercomputing, these computational resources are spread across multiple sites of a grid. Examples of distributed supercomputing applications are numerical simulations[4] and distributed SAT solvers[3]. Supercomputing applications communicate during a computation. This is different from *high throughput* computing such as parameter sweeps, where the computations are performed independently.

Existing grid middlewares do support running supercomputing applications, but have several disadvantages. First, many grid systems lack the ability to simultaneously schedule resources across multiple sites, so called *co-allocation*[5]. Second, setting up and maintaining a grid system can be a challenging and time consuming task. Third, the centralized components present in grid systems hinder fault-tolerance and scalability.

In this paper we will study an alternative to current grid middleware systems: *P2P supercomputing*. In a P2P supercomputing middleware system, no central components of any kind are present, and any part of the system is dispensable. Also, P2P systems are resilient against failures and easy to set up and maintain. This paper especially focuses on one important issue which needs to be addressed before P2P supercomputing can become a viable alternative to grid systems: co-allocation. Current grid systems are able to utilize centralized schedulers when co-allocation is needed. With P2P systems, alternative scheduling algorithms are needed.

For a scheduling algorithm to be useful in a P2P supercomputing environment it needs to have the following properties. First, it needs to be distributed, as central components would compromise the P2P nature of the system. Second, it needs to support co-allocation. Third, it needs to be *locality aware*[9] in that it should schedule jobs with regard for the distance between machines. If a single computation is executed on resources which are too far apart, the communication cost may become too high. Last, the scheduling algorithm needs to be robust against changes in the environment, as P2P systems are highly dynamic.

For the purpose of performing research on P2P supercomputing, we implemented a prototype middleware system: *Zorilla*. Zorilla implements all functionality needed to run applications on a grid in a fully distributed manner. Zorilla was designed to explicitly support supercomputing applications, but it is possible to run any program. It also has support for Ibis[7], a Java grid software system.

Zorilla has a scheduling algorithm which adheres to all the requirements stated above. The algorithm is based on

flooding (a form of selective broadcast). It floods messages over a P2P overlay network to locate available resources. Flooding is done within a certain radius, which influences the number of nodes reached. Zorilla dynamically determines a suitable flood radius.

Using Zorilla, we were able to deploy and run a parallel divide-and-conquer application on 671 processors simultaneously, solving the N-Queens 22 problem in 35 minutes. We used six clusters of the Grid5000[1] system, located at sites across France.

The rest of this paper is organized as follows. Section 2 describes the concept of P2P supercomputing. It also compares P2P supercomputing to related systems. Section 3 presents Zorilla and describes how scheduling is performed in Zorilla. It also describes how Zorilla is implemented. In Section 4 we perform several experiments to see how efficient Zorilla is in scheduling jobs. Finally, we draw conclusions and list future work in Section 5.

2. P2P Supercomputing and Related Work

This section defines the concept of P2P supercomputing as used in this paper. We also describe its relation to grids, global computing systems, and related P2P systems.

The term *P2P supercomputing*, as used in this paper, is used for a system in which a number of *nodes* cooperate in performing computing tasks, or *jobs*. Jobs are typically parallel programs which need to be run simultaneously on multiple machines. The defining property of a P2P supercomputing middleware system is its total lack of central components. All needed middleware functionality such as storage, resource management and communication is implemented in a fully decentralized manner. Any participant of the system is thus able to submit jobs to any node.

The decentralized design of a P2P systems makes it suitable for deployment on system sizes ranging from tens to millions of machines. This paper focuses on a single use case, usually referred to as a *grid*: multiple clusters distributed over a large area connected by a wide area network.

P2P supercomputing differs from *global computing (GC)* systems such as SETI@home[2], XtremWeb[2] and distributed.net[3]. GC systems use spare cycles of workstations to run primarily master-worker type programs. Workers download jobs from a server and upload results when the job is completed. This server is the main difference with P2P supercomputing. In a P2P supercomputing system, jobs are not stored on a server. Instead, storage and scheduling of jobs is performed in a decentralized manner. Another difference between P2P supercomputing and GC systems is that few GC systems support supercomputing applications. If they do (e.g. in the case of XtremWeb) a central scheduler is used to perform the necessary co-allocation.

Another type of system related to P2P supercomputing is grid computing with systems such as Globus and Unicore. These systems typically use compute clusters located at different sites. Every site has a scheduler for the jobs local to the site, and it is possible to submit jobs to remote sites. Co-allocation is usually not directly supported but relies on a centralized meta-scheduler instead.

An advantage of a P2P supercomputing system over grid computing is its ability to automatically adapt to changes in the environment such as nodes joining and nodes failing. P2P supercomputing systems thus require less maintenance than grid systems as new nodes can be added and removed easily. Also, P2P supercomputing systems are more reliable than grids as failing nodes do not compromise the functioning of the system.

There are several problems in P2P supercomputing which are all caused by the lack of central components in the system. The first problem is scheduling. Since there is no central scheduler it is difficult to give any *quality of service (QOS)* guarantees about for instance the number of available resources or the time until a job is started or has finished. Also, all current implementations of schedulers which support co-allocation, such as Koala[5], rely on a central point to be able to make scheduling decisions. In P2P supercomputing a new way of performing co-allocation is required.

Other problems in P2P supercomputing include resource discovery, accounting and authorization. Traditionally, a *Grid Information Server (GIS)* and other centralized information databases are used to implement these services. Distributed implementations are possible, but guarantees about quality of service are difficult.

Also related to Zorilla is Gnutella[4]: a P2P file sharing system. The file search mechanism in Gnutella formed a basis for the resource discovery protocol in Zorilla. In Gnutella, a node sends a search request to its direct neighbors, which in turn send it to all their neighbors. This process continues until a message is forwarded as many times as specified by the user when the search was initiated. The Zorilla search algorithm extends this algorithm in several ways, including dynamically determining a suitable number of times to forward a message, and using the locality awareness of the P2P network to optimize the selection of nodes reached. In Gnutella, a search will reach nodes randomly distributed throughout the network, while in Zorilla nodes are reached that are most likely close to the node where the flood was initiated (see Section 3.2 for more details on the flooding mechanism of Zorilla).

A system closely related to our system is the P2P infrastructure present in ProActive[1] which also strives to use

[1] http://www.grid5000.fr
[2] http://setiathome.ssl.berkeley.edu
[3] http://www.distributed.net
[4] http://rfc-gnutella.sourceforge.net

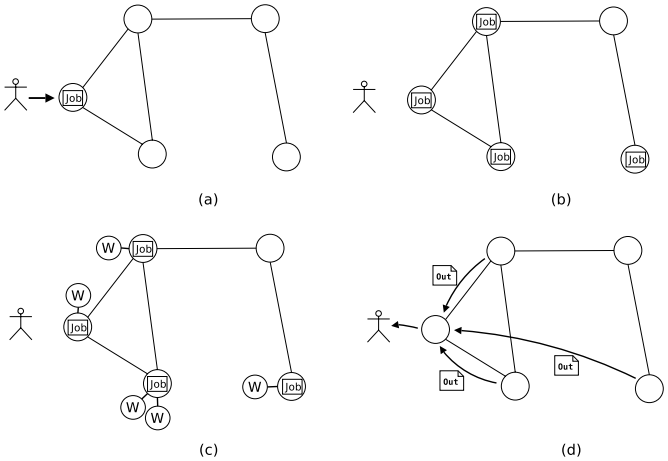

Figure 1. Life of a job in Zorilla.

P2P techniques to perform parallel computations. There are several differences though. ProActive primarily supports applications which use an active object model. Zorilla, on the other hand, supports any application. Another difference lies in the mechanism to discover resources. Like Zorilla, ProActive uses a mechanism derived from the Gnutella system to locate resources, but it extends it differently. For instance, ProActive has no way of dynamically determining a suitable parameter for the *flood* of the resource request. Also, ProActive is not locality aware, as resource selection is based on the load of a remote node, not the distance to other nodes participating in the computation. This approach leads to performance degradation if this mechanism is used to perform P2P supercomputing on machines located across a grid.

3. Zorilla

This section describes Zorilla, a prototype of a P2P supercomputing middleware system. First we describe the system design of Zorilla. Second, we look at how Zorilla deals with the problem of scheduling jobs without relying on any central components. Last, several implementation details of Zorilla are given.

3.1. System Design

The Zorilla system design is based on a network of nodes, or *peers*. A node runs on each machine that is part of the system. As Zorilla is a P2P system, each node has the same functionality. Every node is able to handle job submission, job scheduling, job execution and file storage. Zorilla is completely implemented in Java™ and is portable, as the only requirement put on the system by Zorilla is the availability of a *Java Virtual Machine (JVM)*.

Zorilla nodes are connected by an P2P *overlay network*. An overlay network is a virtual network built on top of a physical network but with added properties such as fault-tolerance and flexibility. An important property of the overlay used in Zorilla is the fact that nodes which are nearby in the physical network are also likely to be nearby in the overlay. Distance is measured in terms of latency between nodes. The locality awareness of the overlay network is used in scheduling jobs, described in Section 3.2.

Jobs in Zorilla are either native or Java applications which need to be run on one or more machines in parallel. Although the Zorilla middleware system is fault tolerant, application level fault tolerance is not provided. When a node fails or is removed from the system, Zorilla will find a new node for the application and notify the application of the new node, but it is up to the application to implement a mechanism to recover from the failure. One possibility is to write the application using a system such as Satin[8], which provides fault tolerance automatically.

When running a job, a user must specify the executable, input and output files of a job, as well as any parameters the application needs. It is also possible to specify the resources (such as memory and number of nodes) the job needs to run.

We now look at the way jobs are handled in Zorilla (see Figure 1). Figure 1(a) shows a job which is submitted to the Zorilla system. Each job is represented by a *distributed Job* object. This object contains the current state of the job and any files the job uses. Each node participating in a computation has a local element, or *constituent*, of the distributed object and is able to enter and retrieve information about the state of the job from this constituent. The initial constituent of the distributed object is created on the node where the job is submitted.

After a job has been submitted to the system it is propagated to as many other Zorilla nodes as requested by the user. The scheduling of jobs is described in Section 3.2. As shown in Figure 1(b), a constituent of the distributed job object is initialized at every node participating in the computation of the job.

When a node has initialized a constituent of the distributed object, it tries to start so called *workers* that execute the user program. Starting a worker might fail if, for instance, the job has already finished since the advertisement was sent, if the number of nodes required in the computation has already been reached, or if the job has been canceled.

In Figure 1(c) these workers are marked with a *W*. A node may start multiple workers, for instance when it has more than one processor. User programs are started in a separate virtual machine, with restricted security settings. These settings enforce the restrictions put on the user program, such as maximum memory usage, and limited access to the node's file system. This security mechanism only works with Java jobs, as the security is enforced by the Java

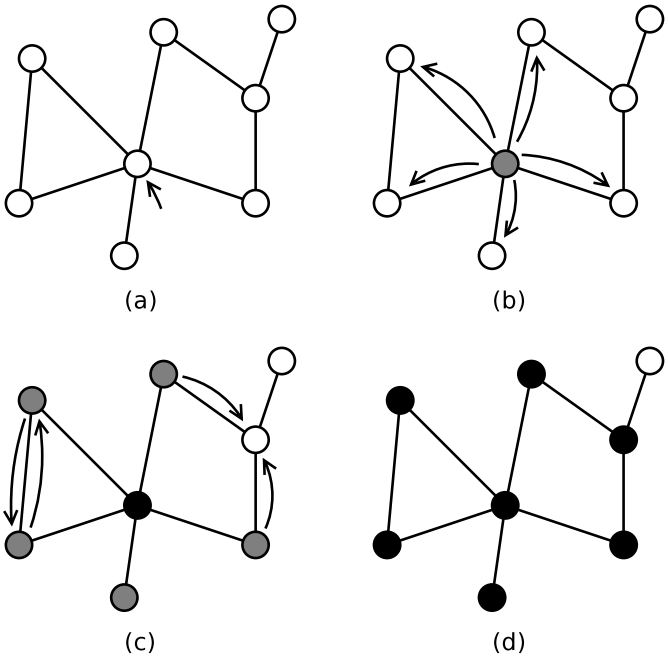

Figure 2. Flooding a message with a radius of two.

Virtual Machine. If security for native programs is desired a virtualization system such as VMware[5] could be used, with only minor modifications to Zorilla.

A running user program is able to read and write the input and output files present in the job. After the user program has exited on all participating nodes, Zorilla copies all the generated output files to the node where the job was submitted, and makes it available to the user who submitted the job. See Figure 1(d).

3.2. Scheduling

Next, we look at the way scheduling is performed in Zorilla. As the scheduling algorithm uses *flooding* to locate resources, we dub this mechanism *flood scheduling*. Scheduling in Zorilla is designed to have the four properties introduced in Section 1. First, Zorilla uses a *decentralized* scheduling algorithm because a centralized scheduling approach would introduce a single point of failure.

Second, the scheduling mechanism supports *co-allocation*[5]. Co-allocation is important when a user wants to run a parallel application, which requires nodes on multiple compute sites concurrently. As Zorilla has no notion of clusters, Zorilla does not schedule nodes on multiple compute sites explicitly. Instead, it treats all the nodes as one big system, enabling usage of nodes in the entire system.

Third, scheduling in Zorilla is locality aware. Nodes participating in a single parallel computation may need to communicate extensively with each other. It is therefore important to keep the distance between the nodes computing a single job as short as possible. Moreover, input and output files need to be copied to and from the node where the job is submitted (the *origin* of a job) and all nodes which perform the computation. Therefore, it is beneficial to select nodes to compute a job from nodes nearby the origin. Zorilla uses the latency between nodes as a measure for distance, and schedules jobs on nodes both near the origin and close to each other.

Last, the scheduling algorithm of Zorilla is robust against changes in the environment. New nodes are automatically considered for running a job and nodes leaving the network do not compromise the functioning of the scheduling.

When a job is submitted to the Zorilla system, an *advertisement* is created for this job. This advertisement is a request to join the computation of the advertised job. It contains information needed to start the job and any resource requirements the job has.

The advertisement for a job is *flooded* by the origin of that job to other nodes in the Zorilla system. In Figure 2, a flood is executed of a message in a Zorilla network consisting of eight nodes. In Figure 2(a), the flood is started at the node marked with an arrow. This origin node then sends the message to all its neighbors in the overlay network in Figure 2(b). The neighbors of the origin node will, in turn, forward the message to all their neighbors, shown in Figure 2(c). In this picture the two nodes on the left forward the message to each other. If a node receives a duplicate message it will be discarded instead of forwarded.

A flood has a certain *radius*, to limit the scope of the flood. The origin of a flood sets the *time to live (TTL)* of the message it sends to the radius of the flood. When a node forwards a flood message to its neighbors it decreases the TTL of the message by one. If the TTL reaches zero it is not forwarded anymore. In Figure 2 the flood is executed with a radius of two. The neighbors forward the message with a TTL of one in Figure 2(c), and these messages are subsequently not forwarded anymore. Figure 2(d) shows the situation after the flood has ended. All nodes with a distance of two or less from the origin of the flood have received the message.

When a node receives an advertisement for a job, it checks if all the requirements of that job are met. Requirements are for instance the availability of processing power, memory and disk space. If enough resources are available, the node initializes a constituent of the distributed object representing the job and starts one or more workers. If it is not possible to run the job at the present time, the node will simply discard the advertisement after forwarding it.

The effect of flooding the advertisement with a certain radius from the origin of a job is that the advertisement is

[5] http://www.vmware.com

delivered to all nodes close to the origin with a certain maximum distance. This maximum distance is dependent on the radius of the flood. A small radius will only reach nodes very close to the node, a large radius will reach more distant nodes as well, and a very large radius might reach all nodes in the entire system. Since the overlay network used in Zorilla is locality aware, nodes nearby in the overlay network are also nearby (in terms of latency) in the physical network. A flood therefore not only reaches the nodes closest to the origin in the overlay network, but these nodes will be physically closest to the origin node as well.

Zorilla uses the following heuristic algorithm to determine the best (as low as possible) radius for the flood that is used to start a job. First, it calculates an initial low estimate. The heuristic currently used by Zorilla is the base 10 logarithm of the number of nodes required, but any conservatively low estimate is sufficient. The origin of the job then initiates a flood of the advertisement with the estimate radius, causing a number of nodes to respond with a request to join the computation. The origin node then checks to see if enough nodes responded to the advert. If not, it will increase the radius used by one, and send a new flood. As nodes usually have in the order of 10 neighbors each, this increase of the radius by one causes the request to be send out to an order of magnitude more nodes than the previous request. Any nodes responding to the new request will be added to the list of nodes in the computation. As long as there are not enough nodes present to perform the computation, it will keep sending floods with increasing radius.

As each flood is done independently, any new nodes which join the network close to the submitting node will automatically receive the request message when the next flood is done. Also, nodes which fail are automatically ignored when a flood is sent as they are no longer part of the network. This makes flood scheduling both flexible and fault-tolerant.

Several optimizations are present in Zorilla to further optimize scheduling. First, if a node is not able to join a computation when it receives an advertisement because of lack of resources, or because it is already running another job, it will not totally discard the advertisement. Instead the advertisement is put in a pending job queue. Periodically, each node will scan its queue and start computing any job which resource requirements are met. To keep nodes from staying in the queue forever, an advertisement comes with an expiration date and is discarded when it expires.

Another optimization is a limitation on the rate at which floods are sent by a node. Since nodes will put the job in their pending job queue if they are not able to start computation, there is a chance some nodes which do not join the computation immediately might do so at a later time. For this reason a node always waits for a certain time before performing another flood. Determining the correct value for this timeout is a problem. If the timeout is too long it will take too long to reach the required number of nodes. If it is too short the radius of the flood will increase too quickly, and will reach nodes far away on the network, while more nearby nodes may still be available.

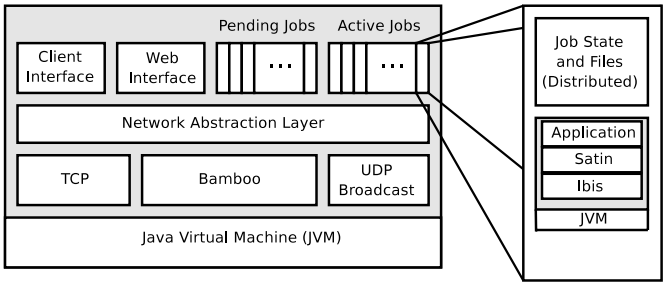

Figure 3. Design of a Zorilla Node.

The solution for the problem of determining the timeout between floods lies in using a flexible timeout. At first, a short timeout is used to reach a reasonable number of nodes quickly. As more floods are send, the timeout increases. The progressively slower rate of sending floods limits the radius and thus the distance to the reached nodes. By default the timeout starts at 1 second and is doubled each time a flood is send, but these values are adjustable if needed.

The last optimization present in the scheduling system of Zorilla is allowing computations to start even when the specified number of nodes is not available yet. Although not all requested resources are there, the limited number of nodes available might still be able to start the computation, if the application supports malleability.

3.3. Implementation

We will now describe the implementation details of Zorilla. Figure 3 shows the layout of a Zorilla node. The Zorilla system design consists of three layers. Zorilla supports three different types of communication, each implemented in the bottom layer. The first communication type supported is TCP, allowing nodes to connect directly to each other for sending large amounts of data efficiently. It also allows external programs to connect to Zorilla, for instance to submit a job.

The second communication type is sending messages via the overlay network. Zorilla uses the overlay network of Bamboo[6]. It enables nodes to communicate with any other node in the system, and allows new nodes to join the system. It also detects node failures, and automatically adjusts the overlay network to compensate. The Bamboo overlay network is also locality aware, as required for the submission system to function properly.

To join the overlay network the address of one or more existing nodes of the network is needed. To find other nodes

the third and last type of communication supported in Zorilla is used: *UDP broadcast*. A node will broadcast UDP packets periodically in the local network to look for other nodes. A user can also specify an address of a peer node if no node can be found in the local network.

The central layer of Zorilla is the network abstraction layer. It hides the specifics of the network implementations to the upper layer, and offers a simple message based system for sending and receiving data.

The top layer of the Zorilla system implements the actual functionality of a node. Job submission can be performed via the *client interface*. It can handle requests made by users via a TCP connection. Possible requests are, among others, submission of new jobs, requesting the status of one or more jobs and cancellation of a job. The state of a node and its jobs can also be monitored via any web browser by means of a *web interface* running on every Zorilla node.

To keep track of which jobs are received by a Zorilla node but not executed yet, each Zorilla node has a *pending job queue*. When an advertisement is received for a job, it is put at the back of the queue. Whenever a Zorilla node is able to start executing a new job, the first job whose resource requirements are met is taken from the queue.

The main data structure of a Zorilla node contains all the jobs it is actively participating in, either because the job was submitted at this node, or because the node is participating in the execution of the job. Each active job has both all the local workers for that job and a constituent of the distributed *Job* object described in Section 3 containing all the files and state of that job. The zoomed part on the right of Figure 3 shows a job with a single worker running an application, in this case a Satin application. See section 4.1 for more information on Satin.

The distributed object can be implemented in various ways. Currently, only a prototype primary/copy implementation of the distributed *Job* object is present in the Zorilla system. In this implementation, the original of the state is kept on the node where the job was submitted (the home node), and other nodes joining have a copy of the state. All calls to the distributed objects which do not alter the state are handled locally on each node. When a call is executed that may alter the state of the job, this call is forwarded to the home node. This node will execute the call on the primary object and distributes any updated state to all copies.

Files are handled differently from the rest of the state. Output and log file data are sent directly to the master copy. Input files are downloaded from a node which is chosen at random, reducing the load on the node where the job was submitted somewhat. This optimization is possible because input files are read only in Zorilla.

4. Experiments

We tested Zorilla on the Grid5000[6] testbed. Grid5000 is a grid system with over a thousand processors distributed over eight sites across France. Most clusters are made up of dual processor AMD Opteron machines connected by a gigabit Ethernet but different processors such as Intel Xeons and IBM PowerPC processors and high speed networks such as Myrinet are also present in the system. Latencies between different sites are 5-20 milliseconds.

4.1. Usability and Scalability Test

To test the usability and scalability of Zorilla we compare running an application on the Grid5000 testbed both with and without Zorilla. The program we used solves the N-Queens puzzle in a distributed fashion and was written specifically for the grid plug-test contest[7], using the Satin system. *Satin*[8] is a parallel divide-and-conquer programming model which allows a user to build efficient, malleable and fault-tolerant applications. In our implementation of the N-Queens puzzle, nodes communicate with all other nodes used, so the application represents the larger class of supercomputing applications, even though the N-Queens puzzle in itself does not.

Satin is built on top of *Ibis*[7], an open source Java grid software system. Ibis enables programs to communicate efficiently with any node in the computation and automatically handles issues such as crossing firewalls . In addition to divide-and-conquer it is possible to write Ibis programs using various other programming models, including standard Java RMI and models which support group communication or message passing.

We first describe the process of running the N-Queens program on the Grid5000 grid system without using Zorilla. Figure 4 lists the steps involved. These steps can be divided into three phases: deployment, running and clean up. Deployment of an application on Grid5000 involves copying the needed files to all clusters and determining the set-up of each cluster involved in the computation. Each cluster in Grid5000 is set-up completely different, so finding the right parameters for the network such as which IP address to use for communicating with other sites is very much a trial and error process.

The second phase, actually running the application, is not so much difficult as it is labor intensive. For each site the number of available nodes must be determined, and the application needs to be run and monitored for failures and errors. As a large number of nodes is used in a single computation, failures are likely to happen. As an extra step specific for Ibis a name server must also be started. This name

[6] http://www.grid5000.fr

[7] http://www.etsi.org/plugtests/History/2005Grid.htm

```
Deployment
    • Copy program and input files to all sites
    • Determine local job scheduling mechanism, write job submission scripts
    • Determine network setup of all clusters
Running
    • Determine site and node availability
    • Start Ibis name server
    • Submit application to scheduler on each site
    • Monitor progress of application
Clean up
    • Gather output and log files
    • Cancel remaining reservations
    • Remove output and log files from sites
```

Figure 4. Running the N-Queens application on Grid5000 without Zorilla

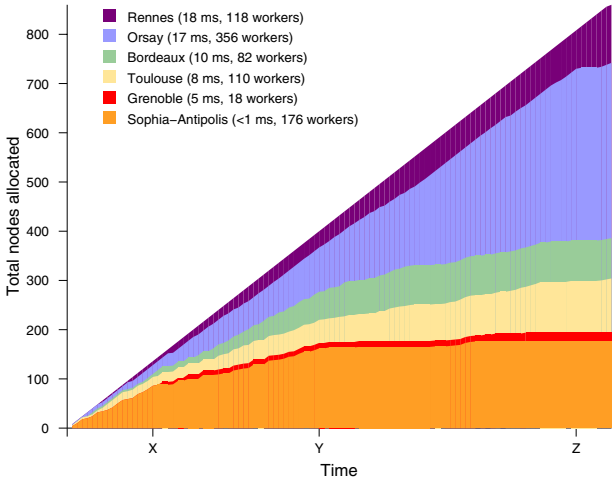

Figure 5. Distribution of workers across sites when submitting an increasing number of jobs from a single node

server acts as an information source for the nodes that are running the application.

The last phase in running the N-Queens application on Grid5000 is the clean up phase. Output and log files are gathered, and any remaining reservations are canceled. Reservation sometimes remain as a result of a failure in either the application or the system.

Next, we ran our N-Queens program with Zorilla. To this end we deployed Zorilla on 338 dual processor machines across six different sites of Grid5000. Each machine ran one Zorilla node with two workers, potentially leading to a total of 676 workers. In a Zorilla grid, Zorilla would be present as a service on all nodes of the grid systems, but for now we ran Zorilla as if it was a normal application. This required us to copy the Zorilla program to all clusters, determine the local network configuration, and start it using the local scheduler.

Once Zorilla was deployed, running the program then involved only a single command to one Zorilla node in the network. Zorilla automatically deployed and started the given program on a total of 671 processors within 90 seconds without any further user intervention. 5 of the workers in the system were not used in the computation due to failures in the nodes, but this did not have any further effect on the computation. After the program was completed, all output and log files were copied back to the node where the job was submitted. Using Zorilla we were able to compute the N-Queens 22 problem in 35 minutes. Of this time under five percent was spend in communication overhead.

Although we started the Zorilla network especially for running only a single application over an existing grid it already improved the usability of the system considerably. Zorilla automated more than half of the steps needed to run a job and made actions such as restarting the application with different parameters much easier and more reliable than would be the case without Zorilla.

4.2. Scheduling

Next, we tested the scheduling algorithm. To this end we deployed Zorilla on a total of 430 machines on six different sites of Grid5000. The maximum number of workers in this system is 860, as each machine again has two workers. The number of workers on each clusters varied, ranging from 18 in Grenoble, to 356 in Orsay.

After deployment we started to continuously submit jobs to the system at one specific node. Each job requested eight workers. As the jobs were set to run forever, the nodes allocated in one job could not be used again in another. Figure 5 shows the distribution of nodes over the different sites over time. The sites are sorted in distance to the submitting node, located at the Sophia-Antipolis site. The graph shows that the clusters closer to the submitting job are allocated before clusters which are located further away.

Up until time X, nodes are almost exclusively allocated out of the Sophia-Antipolis cluster. From time X to time Y, nodes out of the big cluster in Orsay are allocated as well. Between time Y and time Z, almost all nodes of the Sophia-Antipolis cluster are already allocated, and newly allocated nodes mostly are from Orsay. Finally, from time Z on, the Orsay cluster is also virtually allocated, and nodes of the Rennes cluster are used. The graph shows that the flood

scheduler schedules the nodes in a locality-aware fashion.

Although nodes close to the submitting node are *generally* used before far away nodes, the scheduling decisions made are not perfect. Workers are sometimes started on nodes that are not the closest free node. These allocations are a result of the structure of the Bamboo network used in Zorilla. Zorilla floods messages by sending a message to all neighbors of a node. Although Bamboo takes distance into account when selecting neighbors, it cannot *only* use neighbors that are close as this would compromise the functioning of the overlay network. For example, if a network consists of two clusters connected by a very high latency link, selecting only the closest nodes as neighbors would mean no node will ever connect to a node in the other cluster, leading to a partitioning of the network. For this reason the locality awareness of Bamboo cannot be perfect. A node will always have some connections to far away nodes to ensure the functioning of the overlay network. A P2P overlay network which is better suited to support locality-aware floods would be needed to improve the scheduling of Zorilla.

5. Conclusions

We have studied the deployment of distributed supercomputing applications on grids. Because of the limitations of current grid middleware systems (cumbersome to maintain, lack of co-allocation support and low fault-tolerance) we propose P2P supercomputing as an alternative. P2P supercomputing middleware systems are flexible and fault-tolerant. However, new scheduling algorithms are needed as current schedulers rely on centralized components.

We introduced *flood scheduling*: a scheduling algorithm based on flooding messages over a P2P network. Flood scheduling is fully decentralized, flexible and has good fault-tolerance properties. By running a large scale experiment on a real grid (Grid5000) we showed that flood scheduling is able to effectively allocate resources to jobs in a locality-aware way across entire grids. Flood scheduling dependents on the locality-awareness of the P2P network used. Better locality-aware P2P networks are needed if the performance of flood scheduling is to be improved further.

To evaluate our ideas we implemented Zorilla: a prototype P2P supercomputing middleware system. It is easy to deploy, portable, and can automatically perform most of the steps needed to run a distributed supercomputing application. With experiments we also showed that Zorilla has good fault-tolerance and scalability properties.

Future work includes a P2P network specifically designed to support locality-aware flooding, using other measures than latency, e.g. bandwidth, for distance and extending the scheduling system to support economy based scheduling.

Acknowledgments

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). This work has been supported by the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment).

We kindly thank the people of the Grid5000 project, Fabrice Huet in particular, for the opportunity to test Zorilla on their system. We would also like to thank Jason Maassen, Kees van Reeuwijk, Ceriel Jacobs and Gosia Wrzesinska for all their help.

References

[1] J. Bustos-Jimenez, D. Caromel, A. di Costanzo, M. Leyton, and J. M. Piquer. Balancing active objects on a peer to peer infrastructure. In *Proc. of the XXV International Conference of the Chilean Computer Science Society (SCCC 2005)*, Valdivia, Chile, November 2005.

[2] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, and O. Lodygensky. Computing on large scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, 21(3):417–437, March 2005.

[3] W. Chrabakh and R. Wolski. GridSAT: A chaff-based distributed SAT solver for the grid. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing*, page 37, Phoenix, AZ, USA, November 2003.

[4] F. Huet, D. Caromel, and H. E. Bal. A high performance Java middleware with a real application. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 2, Washington, DC, USA, November 2004. IEEE Computer Society.

[5] H. H. Mohamed and D. H. J. Epema. Experiences with the KOALA co-allocating scheduler in multiclusters. In *CCGRID '05: Proc. of the 5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID*, Cardiff, UK, May 2005.

[6] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, pages 127–140, Boston, MA, USA, June 2004.

[7] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an efficient Java-based grid programming environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.

[8] G. Wrzesinska, R. van Nieuwpoort, J. Maassen, and H. E. Bal. Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In *Proc. of 19th International Parallel and Distributed Processing Symposium*, Denver, CA, USA, April 2005.

[9] B. Y. Zhao, A. Joseph, and J. Kubiatowicz. Locality aware mechanisms for large-scale networks. In *Proc. of Workshop on Future Directions in Distributed Comp*, June 2002.