Macro-Level Scheduling in the Cilk Network of Workstations Environment

by

Philip Andrew Lisiecki

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

Author ....
Department of Electrical Engineering and Computer Science
May 24, 1996

Certified by ....
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
F. R. Morganthaler
Chairman, Department Committee on Graduate Theses

Macro-Level Scheduling in the Cilk Network of Workstations Environment

by

Philip Andrew Lisiecki

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 1996, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The term "macro-level scheduling" refers to finding and recruiting idle workstations and allocating them to various adaptively parallel applications. In this thesis, I have designed and implemented a macro-level scheduler for the Cilk Network of Workstations environment. Cilk-NOW provides the "micro-level scheduling" needed to allow programs to be executed adaptively in parallel on an unreliable network of workstations. This macro-level scheduler is designed to be hassle-free and easy to use and customize. It can tolerate network faults, and it can recover from workstation failures. Idleness can be defined in a highly flexible way, in order to minimize the chances of bothering a workstation's owner, but without losing valuable computation time. The security mechanism employed by the macroscheduler can be adapted to make running unauthorized code essentially as difficult as any given system's existing remote execution protocol makes it. This scheduler is also fair, in that it assigns jobs evenly to the set of available processors. Furthermore, the scheduling algorithm utilizes randomness and primarily local information at each processor to distribute the processors among the jobs. Extensive simulations suggest that this algorithm converges to within one processor per job of the fair allocation within $O(\log P)$ steps, where P is the number of processors.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I would like to thank Robert Blumofe and Charles Leiserson. My involvement with this project began several years ago with the Undergraduate Research Opportunities Program helping Bobby with his Cilk-NOW system. My research fairly quickly focused itself on macroscheduling, a component lacking in the original Cilk-NOW implementation. Throughout, Charles and Bobby were great resources for discussing ideas and concerns. Charles has also been instrumental as my thesis supervisor. Charles is exteremely helpful and willing to dedicate time to helping me, even with the inconvenience of his present sabatical in Singapore. I am amazed at the way Charles' simple and concise comments on each draft of my thesis led to such remarkable incremental improvements in the quality of this thesis.

I would also like to thank the members of my family who are financing this whole adventure. They, and the rest of my family, have been extremely generous and supportive. I really can't imagine a better family. Thanks!

Thanks also goes to all of my friends here at M.I.T. and at Random Hall for being there when I've needed them and for being willing to listen to me babbling, as far as they could tell, nonsensically about problems with parts of my thesis. I would like to extend special thanks to my girlfriend, Lorraine Hertzog, who has been extremely supportive throughout, especially when I felt overwhelmed by my thesis.

I have also found W. Richard Stevens' books on UNIX programming [8, 9] invaluable. On several occasions when I just could not solve a UNIX problem, Stevens got me out of my quandary in minutes.

Contents

List of Figures

Chapter 1

Introduction

A network of workstations provides a natural platform for running parallel applications. Since these workstations are unreliable and since workstations are usually intended for primary use by some particular user, these parallel jobs must be adaptive: As resources change, the jobs must be able to expand or contract to match the environment. Empirical evidence shows that network workstations do generally remain idle—typically over half the time—so overcoming the problems of adaptive, secure, fault tolerant, parallel computing is indeed a worthwhile endeavor to provide a cheap, effective pool of computational power. This thesis addresses the issues of running multiple parallel job on such a network of workstations; these issues include fault tolerance, security, fair and efficient distributed scheduling, and detection of idleness.

A natural division of the work of implementing such an adaptively parallel system is to differentiate "macro-level" and "micro-level" scheduling. Here, "macro-level" scheduling, or macroscheduling, refers to recruiting idle workstations and assigning parallel jobs to them. Each job has a "micro-level" scheduler, or microscheduler, that uses this varying set of workstations, assigned by the macroscheduler, to execute its single parallel job. This thesis presents a macroscheduler which was designed and implemented for use with the Cilk Network of Workstations (Cilk-NOW) microscheduler [1, 2].

Other research in scheduling on networks of workstations tends to focus on only one of these two levels of scheduling. Systems such as Piranha [5] focus on running a single job in parallel, using an oversimplified model to balance jobs among processors. Other systems, such as Condor focus only on running single-processor, nonparallel batch jobs on an idle workstation. (See [6] for a comparison of various batch job systems.) The former approach neglects the practicality that multiple users may want to exploit the network of workstations to accomplish their work, while the latter approach fails to utilize a significant portion of idle workstations when the number of jobs is not enough to occupy all of the workstations and generally relies on a centralized scheduler. By combining the Cilk-NOW adaptively parallel, fault-tolerant microscheduler with a distributed, fault-tolerant macroscheduler, Cilk-NOW offers the best of both worlds.

The main features of this macroscheduler are as follows:

Ease of use. The macroscheduler is easy to use. A user can submit a job to the macroscheduler with practically no effort, as if the job were only being run on the local workstation. The Cilk-NOW runtime system automatically submits the job to the macroscheduler, while non-Cilk-NOW jobs can also be submitted explicitly with a simple command-line program.

Fault Tolerance. The macroscheduler can recover from temporary network problems automatically, without requiring users to restart daemons or resubmit jobs to the scheduler. Furthermore, workstation outages do not impede progress on a parallel job. Fault tolerance is also essential to ensure ease of use, as constant user intervention to recover from network and workstation faults is tedious and results in lost computation time.

Flexibility. The macroscheduler is configurable, allowing the conditions used to determine the idleness of workstations to be set dynamically, in accordance with the tastes of the users and the owners of the machines whose cycles are being stolen.

Security. The macroscheduler uses secure protocols that do not open a workstation to unauthorized users running foreign code on a machine. The desired degree of security is that which a given system uses to authenticate its remote execution (`rsh`) protocol. To provide this security, I have created an abstraction on top of active messages [10] called "secure" active messages which offers a quick, point-to-point, secure communication mechanism, which is then used to implement the various protocols in the macroscheduler.

Fairness. The macroscheduler should allocate processors "fairly" among the jobs present. Here, "fairness" means that each job should have an equal share of the processors. The algorithm presented can be extended fairly easily to allow jobs to be assigned relative priorities.

Distributed Scheduling. The scheduling algorithm is random and distributed among the processors, wherever possible. This helps with reliability and fault tolerance, as it limits dependence on centralized, global components of the system. Furthermore, a distributed algorithm could potentially work faster than a global algorithm since a global algorithm must talk to *all* of the processors, while a distributed algorithm likely only needs to talk to a few at a time. Some global information is permissible, however, such as what other processors are present for scheduling at any time, and how many processors are currently involved in each job. Note that this global information may not be perfectly up-to-date, as scheduling decisions are made in parallel, so updated global values take time to propagate to all the processors.

Efficiency. The macroscheduler converges to within one processor per job of the fair allocation in a short period of time, which simulations suggest to be $O(\log P)$, where P is the number of processors. Figure 1-1 shows the convergence of 100 jobs on 10000 processors to the "fair" balance.

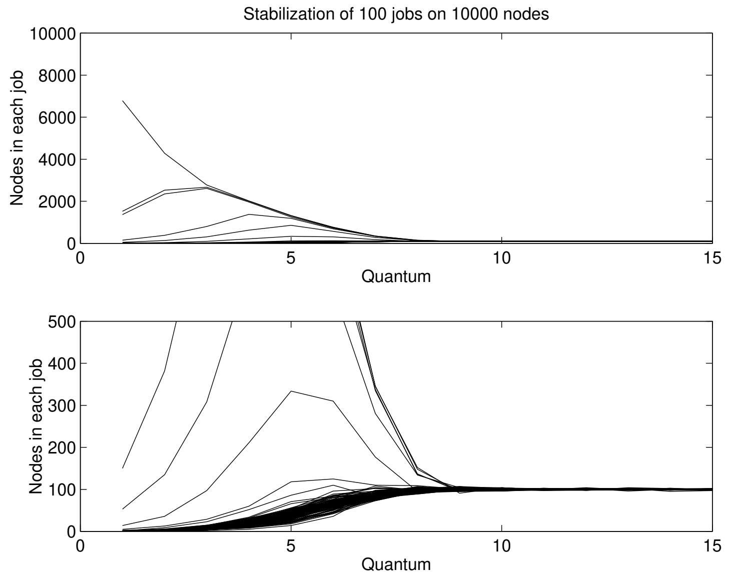


Figure 1-1: Example scheduling run.

In chapter 2, I lay out the basic architecture of the macroscheduler and argue that it gives rise to fault tolerance. In chapter 3, I present secure active messages, the abstraction that guarantees the security of the system. In chapter 4, I present the distributed scheduling algorithm and argue with simulation evidence that it is indeed fair and efficient, i.e., that it achieves an almost-even distribution of processors in $O(\log P)$ time. In chapter 5, I refine the notion of "idleness" and present evidence that networks of workstations provide a sizable and useful computational resource. Finally, in chapter 6, I evaluate the macroscheduler and discuss further work to be done related to this thesis.

Chapter 2

Architecture

The macroscheduler consists of four different types of programs distributed across the network, a job broker, node managers, job managers, and several different utilities.[1] A single machine runs a central "job broker" which is a program called `CilkJobBroker` running as a UNIX daemon at a well-known address on this machine. Each workstation runs a "node manager" daemon which monitors the machine's idleness and spawns or kills workers when appropriate. Each node manager is an instance of the `CilkNodeManager` program. Each job has a "job manager," which is program run on the workstation that starts the computation. The Cilk-NOW runtime system initialization code includes the code necessary to start the `CilkJobManager` program, which acts as that job's job manager. In addition, there are a variety of utility programs (`CilkPs`, `CilkNodes`, `CilkPred`, `xCilkNodeInfo`) that allow the system status and configuration to be displayed and changed. Section 2.1 defines the protocols' assumption that a quick, unreliable, secure way to send messages exists; this mechanism will turn out to be *secure active messages*. Section 2.2 describes the protocols that take a job through its life cycle: node manager registration, job manager registration, scheduling, and job termination. Section 2.3 describes the various utilities that are provided for use with the macroscheduler. Section 2.4 argues that the architecture of the macroscheduler as presented automatically guarantees all facets of fault tolerance with one special exception, which involves a crash of the original worker that started the parallel computation.

[1] Much of the architecture was designed in consultation with Robert Blumofe, now of the University of Texas Department of Computer Science.

2.1 Communication mechanism

Before exploring the basic protocols used by the macroscheduler, some background about the communication mechanism they use is needed. All these programs communicate with each other using protocols implemented with the secure active message abstraction, which is fully defined in chapter 3. For now, these messages can be thought of as simple, quick, unreliable, unforgeable datagrams sent from one program to another; in this section, I discuss the need for speed, unreliability, and unforgeability.

Speed is important, as all of the macroscheduler's operations are performed through communication. Some operations, such as distributed scheduling, require a fairly large number of messages to be sent between large numbers of workstations in order to scheduling jobs. Because several rounds of these messages are required to achieve a fair schedule, the latency of message delivery for each round must be fairly low. For this reason, message delivery should be as quick as reasonably possible, though speed is not as important as for, say, a highly communication-intensive parallel program.

An unreliable protocol is preferable, as using a "reliable" delivery protocol such as TCP/IP serves little purpose, since one cannot guarantee delivery of a message to a node that might fail or across a network which might fail. Since the macroscheduler protocols must be adequately robust to handle any failure gracefully, there is little to gain from a "reliable" protocol; in fact, the overhead of buffering messages that might need to be resent might outweigh the benefits that are reaped from such a strategy. Furthermore, the datagram model is particularly appropriate, since the distributed scheduler requires each workstation to be able to talk to every other workstation, which means a connection-based protocol would either have to track and maintain connections with all the other workstations or else incur the additional overhead of establishing a new connection each time the distributed scheduling algorithm needs to talk to a different node. Using datagrams, the macroscheduler components can merely send a message to any address on the network, and that message will be delivered. Secure active messages themselves are implemented using an active message communication library which in turn uses simple UDP/IP packets to transfer data.

Finally, unforgeability is crucial. Without this assurance from the communication system, an intruder could run unauthorized programs on networked workstations, compromising the network's security. Furthermore, including the security guarantees directly into the communication mechanism ensures that all aspects of the protocol are protected from extraneous or unauthorized messages. Also, by abstracting away the actual underlying security mechanism, secure active messages allow the protocol to operate without significant concern for that security mechanism.

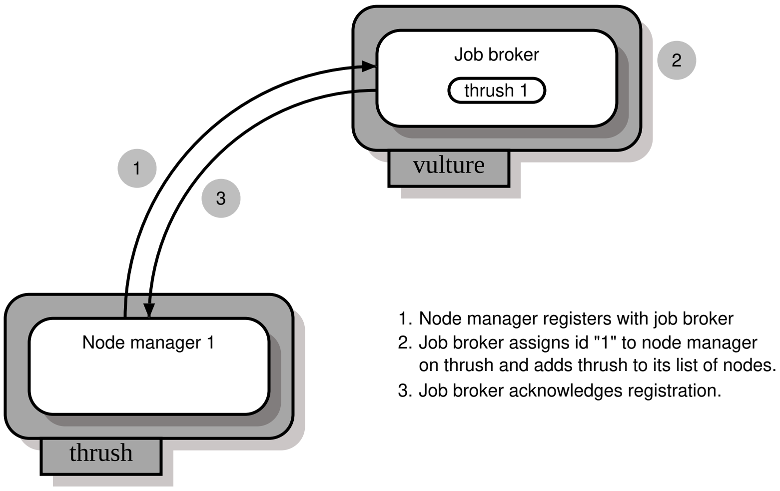


Figure 2-1: Node manager registration.

Hence, these quick, unreliable, unforgeable datagrams are an appropriate form of communication for the macroscheduler's protocols, which will be explored below.

2.2 Life cycle of a typical job

This section presents the life cycle of a typical job. Before a job can be run, node managers must register with the job broker. Once the actual job is started, the job manager must register with the job broker. Then, the job is scheduled. Once the job terminates, the job must be removed from the job broker's list of jobs. Each of these major steps, node manager registration, job manager registration, scheduling, and job termination, is covered in turn below.

2.2.1 Node manager registration

Well before a user wants to run a parallel job, node manager daemons start on all of the machines and register with the job broker so that the job broker will be able to assign jobs to these node managers. In this section, I define the basic node manager registration protocol and explain when the node registers and when it reregisters.

When a node manager daemon starts, say on a machine called `thrush`, it registers its address with the job broker, running for example on a machine named `vulture` (see figure 2-1). The job broker assigns the node manager a unique job-broker-specific *node id* which is used to identify that node for the lifetime of the job broker, and responds with an acknowledgement. Once the node manager is registered, the job broker can begin sending jobs to the node manager to be run on `thrush`. If the node manager fails to register after several attempts, it waits a minute or so and tries to register again. This process repeats until the registration succeeds.

The node manager, once it is registered, goes to sleep, waking up every few minutes to check the idleness of the workstation. This poll rate is increased to once every few seconds whenever a job is running to ensure that when a user returns to his workstation, the parallel job retreats immediately. Whenever the node's status changes, i.e., the node becomes idle, is no longer idle, is now running a job, is no longer running a job, etc., the node manager reregisters with the job broker so that the job broker has reasonably recent information about the node manager's status.

Every few minutes, the node manager attempts to reregister with the job broker by sending it another registration message. Whenever a node manager has not registered with the job broker in some time greater than the registration interval, the job broker assumes the workstation has failed or is no longer reachable on the network and removes the node from its list of nodes that can be used to schedule jobs. If the workstation is actually present, but couldn't get through due to network difficulty, the node reregistration will succeed once the registration requests are again able to be delivered. Hence, if the machine hosting the job broker crashes, only small amounts of computation time might be lost, as the node manager reregisters at some point reasonably soon after the job broker has restarted, though the distributed scheduling protocol to be discussed can often operate even when the job broker is not reachable.

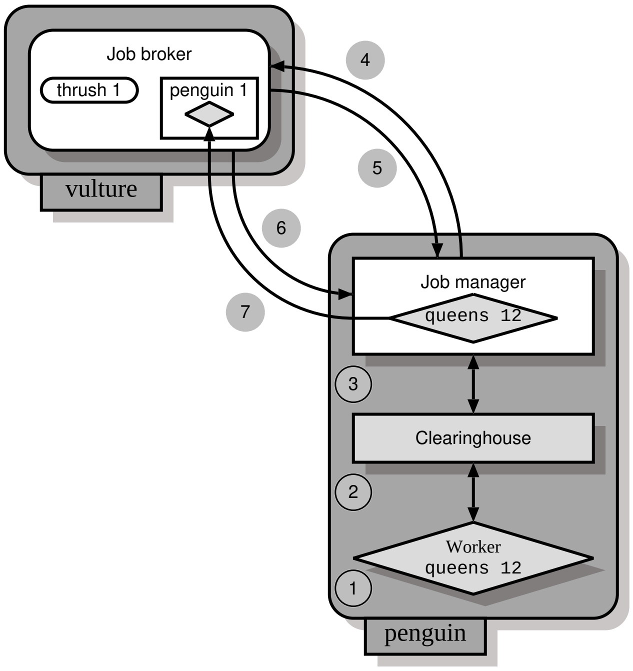
2.2.2 Job manager registration

Once the node managers have registered, a user can begin running a job. When the user starts the job, a job manager will be started which will register with the job broker, allowing scheduling to begin; the details of this job manager registration are discussed below.

For example, a user on a machine called `penguin` can run a parallel program with a simple command line such as

```
queens 12
```

to begin the N-queens program on a 12 by 12 board (see figure 2-2). The Cilk-NOW runtime system starts up a special process on `penguin` known as the *clearinghouse*, an instance of the supplied `CilkChouse` program. This clearinghouse acts as a central hub for information pertaining to the participants in a given parallel job, and hence is part of the microscheduler. This clearinghouse, in turn, begins the macroscheduling of the job by spawning a job manager on `penguin`, passing it as a



1. User starts a job by typing: `queens 12`
2. The new worker spawns a clearinghouse and registers with it.
3. Job manager sends registration request to job broker.
4. Job broker assigns job id "1" to job manager on penguin and adds the job to its list of jobs.
5. The clearing house spawns a job manager, which launches the macroscheduling of the job.
6. Job broker requests the actual job information.
7. Job manager sends the job information.

Figure 2-2: Job manager registration.

command line argument the command line to start other workers. For this example, the command line might be

`CilkJobManager -- queens -NoChouse -Host=penguin -Port=`*clearinghouse-port* `--`

Everything after the first "`--`" is part of the command line to be run by subordinate workers; specifically, it instructs the new workers not to start their own clearinghouse, but instead to use the one found on `penguin` at the specified port.

When this job manager is started, it registers itself with the job broker by telling the job broker at what address the job manager is running. The job broker adds that job manager to its list of all jobs that can be run. Upon receiving the registration request, the job broker assigns the job a *job id* which is then used to identify that job for the lifetime of that job broker and acknowledges the registration request, including the new job id. The job broker still does not have the actual job information for the job; to obtain this information, the job broker sends a request to the job

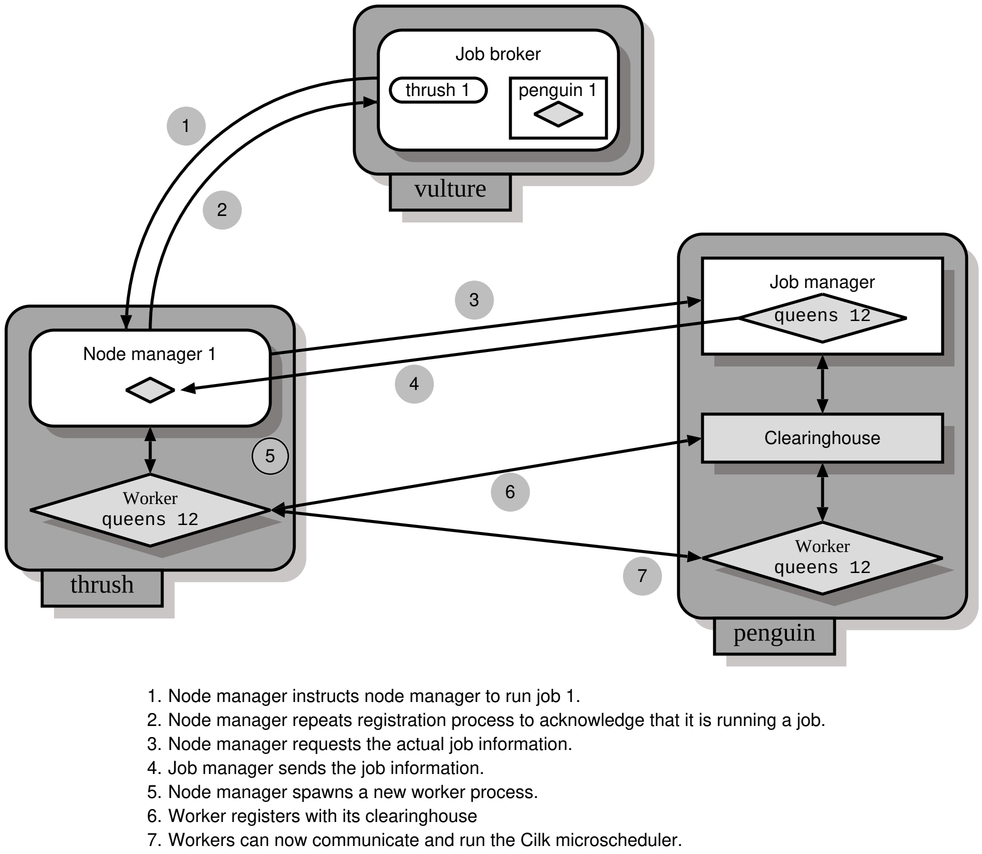


Figure 2-3: Global macroscheduling.

manager which responds with the job information, which includes the command line of the program that will be the worker, information about the user running the program, the UNIX environment to be established for the workers, and the current working directory. Now scheduling of the job can commence.

2.2.3 Scheduling

Once the job broker has jobs in its job list and nodes in its node list, it can begin the scheduling process. This process is catapulted by a *global* scheduling operation, and completed by *distributed* scheduling, each of which will be discussed below.

The job broker is responsible for global scheduling. (See figure 2-3.) To perform global scheduling, the job broker issues a command to a single node manager that identifies a job id and the address of a job manager to begin running that job.[2] As long as each job is assigned to at least one workstation, any imbalance in the jobs is corrected by *distributed* scheduling operations.

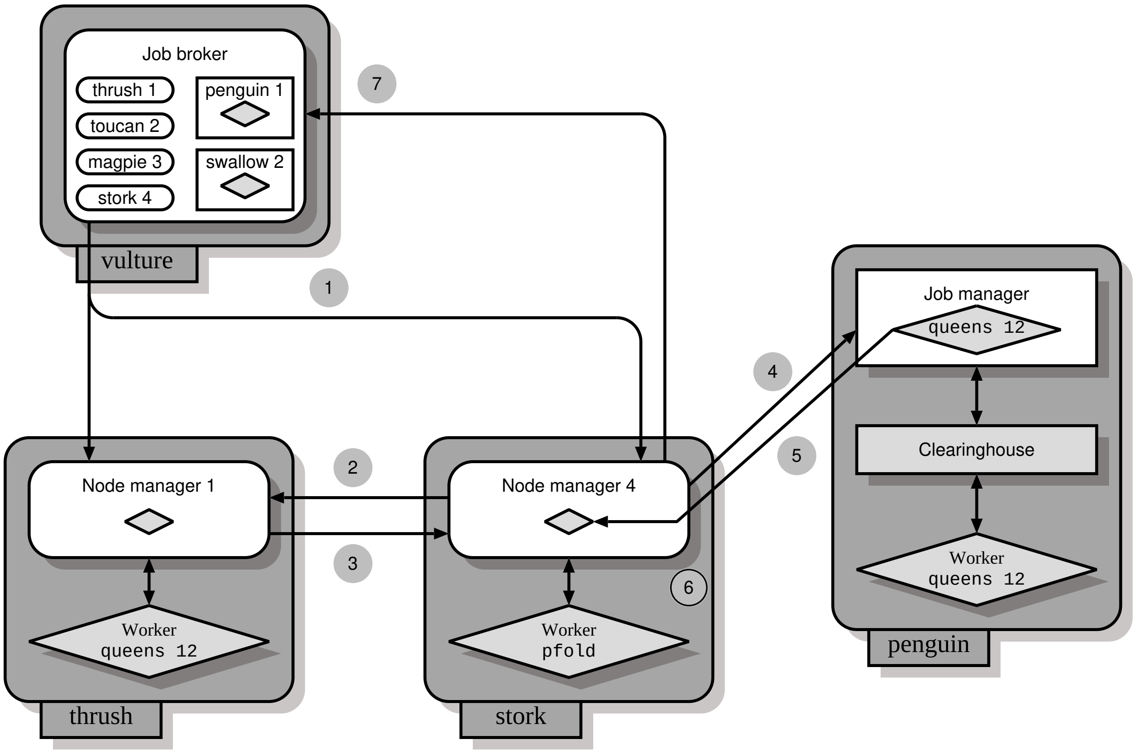
The actual information about the job is retrieved by the node manager directly from the job manager. Since only the job manager is trusted to represent the user, this architecture allows stronger security claims to be achieved, as described in chapter 3.

Once the job information is received, the node manager on `thrush` spawns a new worker, passing in the command line specified above. This worker's microscheduler then registers with the job's clearinghouse, which in turn informs the new worker of the other workers for that job. The microscheduler can then start scheduling work on the new worker.

The distributed macroscheduling protocol operates between pairs of nodes, say for the sake of illustration, `thrush` and `stork`. (See figure 2-4.) The actual distributed macroscheduling protocol involves random pairs of node managers contacting each other and possibly changing jobs. (The full details of when and how these pairings occur is detailed in chapter 4; this section focuses on the protocols that actually perform the pairing and, possibly, switch jobs.) In order for node managers to find other node managers, they must know where the other idle node managers are on the network; the job broker provides updates to the node managers whenever workstations become idle or busy, so the node managers can maintain a list of idle workstations and their node managers' addresses. Also, the node managers must have estimates of the number of workers participating in the local worker's job. This information comes from two places: First, the job broker updates the node managers periodically when node counts change, and second, the workers can send their worker count to the node manager, as the Cilk-NOW microscheduler knows this number, as well.[3] Of course, all this information is typically a little out of date as jobs are started and stopped all over the network in an unsynchronized fashion. When a node manager wants to pair with another node manager, it picks a random node manager from its list of node managers. A message is then sent requesting the remote node count, job manager address, and job id; the other node manager replies with this information. Then, based on the distributed scheduling algorithm of chapter 4, the node manager may choose to switch jobs. If this switch happens, the node manager contacts the specified job manager to get the actual job information for the job and also reregisters with the job broker to indicate that it is now

[2] In order to save network traffic when the system is completely idle, if no job is running on a registered node manager, it does not seek jobs by distributed scheduling; for this reason, the *first* job started is actually started by the job broker on every node manager that is available to run jobs. When a node manager is not registered, i.e., it has not received a registration acknowledgement recently, it begins the distributed scheduling process to look for jobs.

[3] Having the job broker as well as workers provide node counts allows jobs that do not provide node counts to their node managers to be scheduled using the macroscheduler whiel the job broker is operational, if need be.



1. The job broker informs the job brokers of a recent count of nodes working on a particular job when the count changes. The job broker similarly informs the job brokers of the addresses of node managers on recently idle workstations.
2. The node manager on stork queries the node manager on thrush as to what job is being run and how many nodes are working on that job.
3. Node manager 1 answers the query.
4. If the scheduler decides to switch jobs, the new job is requested from the job manager.
5. Job manager sends the job information.
6. The old worker is now killed and the new one is spawned.
7. The node manager reregisters with the job broker to indicate that it has changed jobs.

Figure 2-4: Distributed macroscheduling.

working on a different job.

When a node manager determines that a job change is necessary or that the workstation is no longer available for parallel computations, it must stop the current worker on that node. The node manager first sends a UNIX quit signal to the worker, requesting that it terminate. If the process does not terminate within a minute, it tries again; if this too fails, the node manager kills the process forcibly with the uncatchable `SIGKILL` signal. The Cilk-NOW microscheduler automatically catches the quit signal and migrates all of its work in progress to other workers, so no work is lost when a worker terminates. Meanwhile, the node manager tells the job broker that it is no longer available for scheduling computations by reregistering the node manager with its updated state information.

2.2.4 Job termination

The final step in the life cycle of a job is termination of the job. When a job finishes executing, the job manager detects that the job's clearinghouse has terminated and then unregisters the job from the job broker using a protocol which is almost identical to the original registration, and then the job manager terminates. The global and/or distributed macroschedulers then reassign jobs to rebalance them among the nodes.

2.3 Utilities

Various utilities communicate with the job broker and provide a variety of useful functions:

CilkPs allows users to list the current jobs being scheduled. This utility is analogous to the UNIX `ps` utility.

CilkNodes allows users to list all of the nodes currently known to the job broker, as well as their current status (not idle, running a particular job, waiting for a job to run, etc.).

CilkPred allows users to examine and modify the conditions, or "predicates," used to specify idleness.

xCilkNodeInfo allows users to open an X-window display of activity on a given node.

These utilities are all implemented using an interface consisting of a variety of useful services provided by the job broker. The protocols that these services use are extremely simple, as they generally only require the job broker to gather or change information and then respond to the utility. Viewing or updating idleness conditions is a little more involved, since it can require contacting a node manager to gather information and then echoing the resulting information back to the utility. These utilities provide the mechanisms needed to administer and monitor the macroscheduler's operation.

2.4 Fault tolerance

The architecture of the system and the protocols described above automatically provide the framework for the system's fault tolerance, with one exception involving the machine hosting the job manager. In this section, I argue that indeed the above protocols address the problems of fault tolerance, and then I suggest an additional protocol to fix the one exception.

Generally, recovery from workstation failures and temporary network failures is as simple as reregistering after the failure has ended. If a machine hosting a node manager crashes, the worker on that machine will be lost; the Cilk-NOW microscheduler, however, is able to tolerate such a fault. Once the node manager restarts, it will reregister and begin running jobs again; in the mean time, the other node managers will rebalance the jobs among the remaining nodes. Should the network become unavailable temporarily, all components will cease to function, as communication is essential to a parallel system; however, when the network becomes operational again, all of the job managers and node managers will reregister, and proper scheduling will continue almost immediately.

The only workstation, other than the one hosting the job manager, that can stop a job from being scheduled if it is down for prolonged periods of time is the job broker, but distributed scheduling affords a bit of leeway here. Without the job broker, new jobs cannot be scheduled, and newly idle nodes cannot receive jobs, though as long as a job is running somewhere and nodes stay available, it continues to be scheduled.[4]

Hence, as long as job broker outages are not permanent and the "root" worker (i.e., the original worker that spawned the job manager) of the computation stays up, a job continues to be scheduled on the network of workstations. This makes the parallel execution of a program essentially as reliable as the microscheduler being macroscheduled.

Unfortunately, if the machine hosting the root of the computation crashes, all traces of the computation, as managed by the above protocols, are lost. The Cilk-NOW microscheduler has support for recovering a computation whose root participant and/or clearinghouse was abnormally terminated using *checkpoint* files written periodically during the computation. In order to recover, however, a new job manager must be started on the same node as the original root was started (for security reasons, among others). Hence, whenever a job manager registers, the job broker must instruct a node manager on the same node as the root of the computation receive a "backup" copy of all the job information in case recovery is necessary; this information is stored in persistent storage, such as in a file on the local hard drive, so that the information survives a workstation crash. Whenever a node manager registers freshly, either because it or the job broker has restarted, the node manager must inform the job broker of all of the pending "crashed" jobs. When the job broker deems it appropriate, it must send a special command to the node manager to spawn a new job manager, which then registers the crashed job and starts up a new root worker, instructing it to

[4] In order for jobs to continue to be scheduled, nodes must not switch between being idle and being busy too frequently. This is generally the case, as described in section 5.3.

recover the checkpoint files. This procedure is not currently implemented in the macroscheduler.

In this way, the architecture of the system naturally encompasses all but one concern of fault tolerance, namely, recovery from the crash of a root worker or clearinghouse; the system, however, can be adapted with only a small amount of added complexity to support automatic checkpoint recovery.

Chapter 3

Security

This chapter defines the security requirements of the macroscheduler and discusses how they are addressed. In the first section, I present these security requirements and the "secure" active message protocol which I created to address these concerns. The key issue to security is that the macroscheduler allows UNIX programs to be run remotely across the network, so the security must be at least as strong as a system's remote execution protocol. Then, I consider the implications that this security model has on the architecture of the macroscheduler; in particular, security concerns force the job manager to provide *all* node managers *independently* and *directly* with all job information. Finally, I present an implementational variation on secure active messages, the secure active message server, that eases administration and strengthens security in some cases.

3.1 Security model

The security requirements of the macroscheduler are quite similar to those of a remote execution protocol: The macroscheduler allows programs originally started on a user's local machine to be run on remote computers. For this reason, the macroscheduler must be at least as secure as the machine's remote execution (`rsh`) protocol. Because different installations use different protocols to administer remote execution, the macroscheduler must be easily adaptable to a variety of different authentication schemes, such as the standard UNIX `rsh` protocol [8, chs 9&14] or Kerberos [7]. Different authentication schemes present different demands and use different fundamental security models. Encapsulating the actual authentication protocols in a simple abstraction and then using the strictest assumptions used by any of the authentication protocols in the design of the macroscheduler ensure that the macroscheduler is secure and yet not overwhelmed with the complexity of authentication and authorization; the secure active message abstraction captures this notion.

3.1.1 Secure active messages

Secure active messages provide a highly efficient, unreliable (i.e., no guarantee of delivery) messaging system that fulfills these fairly rigorous security requirements. The sender and receiver of secure active messages are not just network addresses, but instead *principals*, which encompass a network address and a claim as to the identity of the sender. The receiver of any secure active message that is successfully delivered is provided with the identity of the sending principal, as well as additional authentication information about the message. Each secure active message also contains information as to which *handler* on the remote machine should receive the message. In this section, I describe how secure active messages are sent and received and describe the cautions to be observed when using them to ensure a secure system.

Speed is fairly important, as the distributed scheduling algorithm requires nodes to pair up with many other nodes. The actual remote execution utilities whose security is to be emulated (i.e., `rsh`, `rcmd`) are typically fairly slow to use directly, and they cannot be made to perform distributed scheduling securely and efficiently, as each communication path between workstations would require its own secure remote execution channel. For this reason, a leaner secure communication system is needed, and secure active messages fulfill this need.

Because the requirements of secure active messages are quite similar to those of active messages but with added security, the secure active message layer is actually implemented as an additional layer above an existing active message implementation.[1] An active message system encapsulates in each message the address of a procedure to execute on the remote node and data that is passed to that procedure. Theoretically, such a protocol could be implemented with a minimum of buffer copying, etc., providing an extremely low overhead messaging system.[2] [10]

In order to make active messages secure, function addresses cannot be passed directly over the network without being confirmed for correctness at the receiving end. Instead of incurring this

[1] Actually, there is an intervening layer which splits long messages into many active messages and sends them to the remote host, where they are reassembled, and presented to the secure active message layer as if they were a single active message. Small messages are sent directly as active messages, and so do not incur any overhead beyond a size comparison. This layering is reminiscent of that used by the Strata communications library [4].

[2] The active message implementation utilized by the macroscheduler works on top of UNIX's socket interface, and so does not achieve the maximal theoretical throughput.

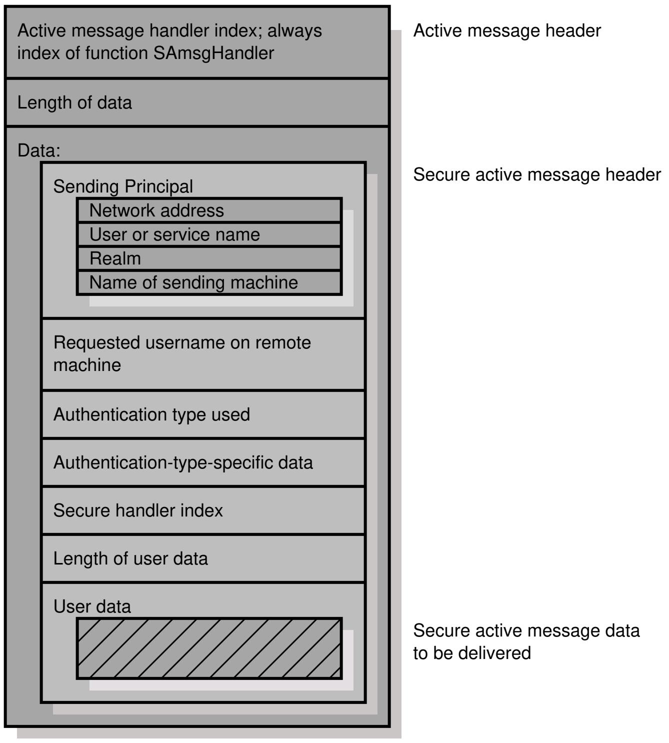


Figure 3-1: Secure active message format, including the active message header which wraps the secure active message data.

verification overhead on each message, and realizing that different workstations might not load a program at the same address, or even that the sending and receiving programs may be different programs altogether, these addresses are replaced by indices into a *handler table*. This handler table then contains all of the functions that have been specifically registered as active message *handlers*.

To provide actual authentication, all secure active messages are sent to a special active message handler that receives the active messages and performs the appropriate authentication on them. A special header before the original message contains information about which authentication scheme (UNIX `rsh`, Kerberos, etc.) is being used on this message, who the sending principal is, what remote authentication is being sought, and which handler should be invoked when authentication is complete, as well as any authentication scheme specific data (see figure 3-1). The handlers that are invoked once a secure active message has passed the authentication step are chosen from a second handler table, since these functions need additional arguments to convey information about the sending principal and the authentication type used. Only messages which pass the authentication step get delivered to the handler; other messages are ignored.

Although all secure active messages have passed an authentication step, the actual authorization needed to run a given handler must be checked by the handler itself, allowing a handler's authorization policy to be arbitrarily complicated. Each handler must therefore ensure that the sender is authorized to perform the requested service, and that the authentication method used was adequate for that service. When the sending principal is already known, a simple routine (`SAmsgVerifySender`) allows a simple one-line check to ensure that the sender is the expected sender.

The secure active message system does not make any guarantee about the contents of the secure active message, other than that they are what the sending principal intended them to be. For maximal security, all handlers should assume that the remote party is malicious and not let invalid or missing data compromise the operation or security of the program.

3.2 Architectural implications

When an actual job gets run, the node manager should know that the job to be run is actually authorized by the user that wants to run the program; if the node manager does not have a solid assurance of the user's identity, an intruder could start an unauthorized process on a machine, allowing him to view, modify, or destroy the user's data. In this section, I argue that the job manager's design maintains this authorization assurance for even the the strongest of security primitives.

In the case of standard UNIX `rsh` authentication, the system architecture is not very restricted, as all components must *fully* trust all other components that are running on trusted machines. However, with a system such as Kerberos, the superuser on any machine cannot be trusted to represent a user; only a user that possesses tickets should have this authority.

For this reason, whenever the job broker or a node manager needs information about a job, it gets that information directly from the job manager. This is not necessary for a system such as UNIX `rsh` but is essential to keep the security model of a system such as Kerberos intact. Because the job authentication is always directly from the job manager to the node manager, the authentication scheme is essentially equivalent to that used by the Kerberos version of `rsh`.

Another nice benefit of always getting job information from the job manager is that if additional authentication information needs to be transferred to the remote machines in order to give them access to file systems, etc., the job manager could be easily modified to transfer this additional data directly to the node managers with each job request.

3.3 Secure active message server

Some authentication schemes prevent an ordinary user program from performing authentication. In this section, I explore the problems associated with this restriction and present an implementational variation on secure active messages, the "secure active message server," that addresses these concerns.

As an example of this problem, the UNIX `rsh` protocol requires superuser access to bind to a "reserved" port. Each program utilizing secure active messages would then need to be tagged "setuid root" by the system administrator. Furthermore, all code that uses secure active messages must then be thoroughly trusted not to abuse root access to the workstation. In fact, the problem is even more severe, as a program which has root access in the UNIX `rsh` scheme has universal access as all other principals on all other machines.

For these reasons, a drop-in replacement for the secure active message library allows the actual secure active message system to be run as a server in a separate process. Using this server, users can write utilities that talk to the various components of the macroscheduler without the extreme difficulty of having to get their programs entrusted with superuser privileges. Furthermore, less code needs to be trusted when writing utilities, as the utility itself runs without the additional permissions needed to perform authentication. Of course, authentication schemes which do not require special permissions, such as Kerberos, may bypass this extra unneeded layer of protection by linking directly against the actual secure active message library.

Chapter 4

Scheduling

The scheduling subsystem of the macroscheduler ensures that the processors (i.e., workstations represented by node managers) are allocated fairly among all the jobs.[1] The schedule sought is one with a static allocation of jobs to processors, so the number of processors, P, must exceed the number of jobs, J. An underlying assumption of this objective is that the jobs represent independent tasks that require large numbers of processors each. Like a round-robin time-shared environment such as UNIX, the objective is to give each user an equal share of computation time. Issues such as scarce resource sharing, often considered a part of scheduling problems, are not a concern, as each workstation has approximately equivalent resources. Furthermore, as jobs are independent, there is no "critical path" to be optimized. The objective of this scheduler is to provide a simple, very-low-computation algorithm to achieve this even split quickly and efficiently. As a practical concern, jobs may be given relative priorities; the given algorithm can be extended to account for these priorities. Furthermore, if some jobs do not need all of their even share, the excess processors should be distributed among the jobs that can utilize them; this issue is not addressed here, but some ideas are given in chapter 6.

The actual macroscheduler is divided into two parts, a minimal centralized global macroscheduler run by the job broker and a distributed macroscheduler run by all of the node managers. The global scheduler provides the mechanism to start jobs and change the set of idle workstations; the distributed scheduler provides an efficient, fair scheduling of the jobs. Essentially, the global

[1]The following people have made very significant contributions to the scheduling algorithm: Charles Leiserson of the M.I.T. Laboratory for Computer Science, Robert Blumofe and David Zuckerman of the University of Texas Department of Computer Science, and Aravind Srinivasan of the National University of Singapore.

macroscheduler watches what processors are running what jobs, and if it notices a job with no processors, it assigns some processor to that job. This is the mechanism by which a newly submitted job first gets started. Furthermore, if all the processors running a particular job happen to go down, the job broker detects that the job is no longer being run and assigns it a processor. The mechanism used by the global macroscheduler to change the active job is indicated in section 2.2.3. The distributed macroscheduler then balances all of the jobs fairly among all the workstations available; this distributed macroscheduler is the focus of the remainder of this chapter.

The distributed macroscheduler uses a simple, randomized algorithm to balance processors between jobs. In the following sections, I shall argue the basic structure of the distributed scheduler and present an analytical justification for its form. Then, I argue on the basis of simulation evidence that this algorithm achieves a near-perfect allocation (to within one) in $O(\log P)$ time.

4.1 Distributed macroscheduler form

In this section, I discuss several alternatives for a randomized distributed scheduler and argue that the form used by the macroscheduler, unlike the others discussed, is a viable alternative for efficient, fair scheduling.

The fundamental idea is to have each processor periodically pick another processor at random and see whether that processor's job is more in need of resources than the local job. If so, the job may be switched, depending on the outcome of some locally run algorithm. A predefined (but unsynchronized) interval that is used to determine when to perform a scheduling step and is called a *quantum*; each processor engages in the scheduling algorithm once in each of these quanta.

The simplest imaginable switching policy is to switch whenever the other processor's job is more in need of resources. Unfortunately, the naivete of this scheme backfires rather severely, since the processor counts may be up to a quantum out of date: So many processors switch jobs that the imbalance of processors switches polarity, often leaving the magnitude of the mismatch greater than it was to start. Simulations of this strategy reveal that it is completely unstable, often to the point that one job ends up with all the processors within a small number of quanta.

A slightly more complicated switching policy, but still inadequate, would be to switch when an imbalance is seen with some predefined fixed probability, or with some probability determined by the total number of processors and/or the total number of jobs. The former strategy fails as the size of the scheduling problem that can be handled is determined by the value of the constant, while the latter requires more global information, and does not achieve equilibrium quickly, as the constant must be low enough that overswitching is unlikely even when two processors are almost balanced; hence, the imbalance is taken up at most about one processor per quantum, meaning this algorithm is $\Omega(P)$.

The algorithm employed by the distributed macroscheduler is quite similar to the previous one, except that the switching probabilities are determined by some function of a recent processor count of the local job and a recent processor count of the remote job. The following derivation suggests such a function, which simulations according to simulations achieves the desired almost-fair schedule in $O(\log P)$ quanta.

4.2 Derivation

In this section, I present an analytic derivation for a randomized macroscheduler algorithm using the switching policy just mentioned for the case of two jobs; this algorithm attempts to make maximal progress toward equilibrium at all times without a high risk of overshooting that equilibrium. Then, I argue on the basis of simulation evidence that the behavior for three or more jobs is still within the desired $O(\log P)$ quanta.

The actual algorithm is derived for the case of only two jobs ($J = 2$), job A and job B, and then later extended to many jobs. At any time, job A has N_A processors and job B has $N_B = P - N_A$ processors. For any processor p, denote the number of processors working on the same job as p as $N(p)$. To model the delay in processor count updates, the algorithm uses a processor count denoted $m(p)$ which is equal to $N(p)$ at the beginning of the quantum. Since any processors p and q working on the same job C has $m(p) = m(q)$, we define $m_C \equiv m(p) = m(q)$. Without loss of generality, assume $N_A < N_B$ at the beginning of a quantum, so $m_A < m_B$. Hence, only processors running job B consider switching jobs during this quantum. The pairings made by processors running job A then do not affect the number of switches made, since for any job q, $m(q) \geq m_A$. The remaining m_B processors running job B each pick a random partner, which is running job A with probability N_A/P. If processors that switch from job B to job A do not engage in further pairings for the rest of the quantum, this probability becomes m_A/P. The expected number of processors, $\mathit{expected}(B, A)$, running job B that find a processor running job A (which can then induce a job switch) is

$$\mathit{expected}(B, A) = \frac{m_A \cdot m_B}{P}.$$

The requirement that nodes stop pairing is fulfilled in the actual macroscheduler by the fact that a node manager that has just switched jobs refuses distributed scheduling requests until it receives an initial node count from the job broker or the worker.

The desired even split is $N_A = N_B = P/2$. Since each switch closes the gap between N_A and N_B by two processors, the *desired* number of switches, $\mathit{desired}(B, A)$, is

$$\mathit{desired}(B, A) = \frac{m_B - m_A}{2}.$$

Of all of the $\mathit{expected}(B, A)$ pairings, $\mathit{desired}(B, A)$ of them should succeed. If each processor switches with probability

$$\begin{aligned} \rho &= \frac{\mathit{desired}(B, A)}{\mathit{expected}(B, A)} \\ &= \frac{(m_B - m_A)P}{2 \cdot m_A \cdot m_B} \\ &= \frac{(m_B - m_A)(m_B + m_A)}{2m_A m_B} \\ &= \frac{m_B^2 - m_A^2}{2m_A m_B}, \end{aligned}$$

then, as long as $\rho \leq 1$, the expected number of switches is $\mathit{switches}(B, A) = \rho \cdot \mathit{expected}(B, A) = \mathit{desired}(B, A)$, the appropriate number to close the gap. Note that ρ may be greater than one; in this case, all processors presented with the opportunity to switch do so; this behavior makes maximal progress toward equilibrium, without being in danger of overshooting it. When $\rho \geq 1$, $\mathit{switches}(B, A) = \mathit{expected}(B, A) = m_B m_A / P$. If we assume $N_A \ll N_B$, then $\rho > 1$ and $m_B/P \approx 1$, so $\mathit{switches}(B, A) \approx m_A$. Hence, while $N_A \ll N_B$, the number of processors running job A is expected to double in each quantum. Without proof, it makes sense that the expected order of growth of this algorithm is $O(\log P)$ for the $J = 2$ case. (The proof must consider what happens when $N_A \ll N_B$ does not apply but $\rho > 1$. One must also consider the actual distribution of switches and the effect of possible overshoot in the distribution when more processors than expected decide to switch jobs.)

When there are many jobs ($J \geq 2$), applying this same algorithm still works pretty well, until all processors are within one processor of the even allocation. Each processor just picks another processor in each quantum, and compares the number of processors participating in each of their jobs; no other jobs are considered when computing a switch probability. As long as a single job i is outside this range, there is guaranteed to be an extremely high probability in each quantum of finding a job j such that $|N_i - N_j| \geq 2$, so swift progress is made toward equilibrium. The general behavior observed is that the job with the most processors and the job with the fewest processors tend to make reasonable progress toward equilibrium while the intermediate jobs tend to stay sandwiched between these two jobs. However, when all jobs are within one of equilibrium, in order to make progress, a job one over equilibrium must find a job one under. In the worst case, all jobs but two are in equilibrium with $P = 2J$, so $N_A = 1$, $N_B = 3$, $N_i = 2\ \forall i \neq A, B$. In this case, job B finds job A with probability approximately $(N_B N_A)/P = 3/(2J) = O(1/J)$, and hence achieving perfect equilibrium takes $\Omega(J)$ time in this case.

Figure 4-1 shows the stabilization of 100 jobs on 10000 processors. Note that the job with the most processors falls off very rapidly, and the jobs with the fewest processors rise in an exponential fashion to the equilibrium when they are far away. The intermediate jobs are pulled away from equilibrium at first, but as soon as they lie near the extreme, they begin falling off with the maximal job. In this example, all jobs are within one of the fair distribution by the sixteenth quantum.

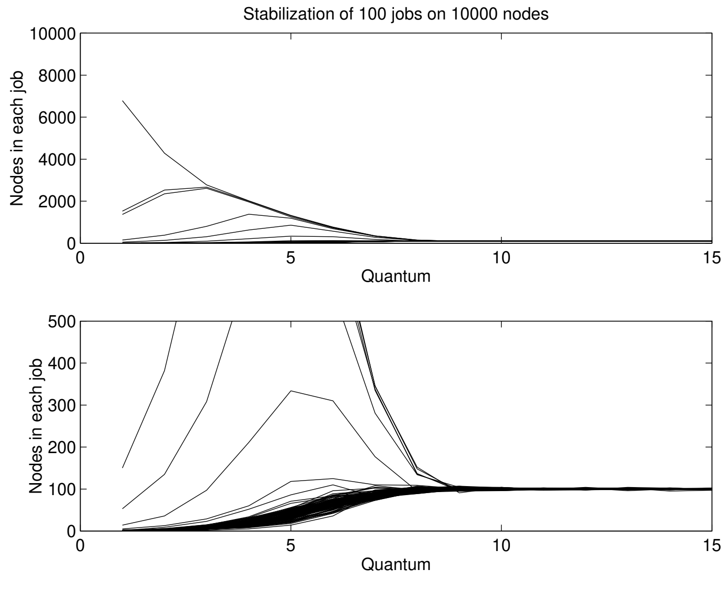


Figure 4-1: Example scheduling run.

4.3 Simulation results

Extensive simulations suggest that this algorithm is efficient and stable. The stabilization for any number of processors and any number of jobs strongly resembles the behavior exhibited by figure 4-1 and described in the previous section. Numerous simulations with different numbers of processors and jobs were run; the average number of quanta required to reach the almost-fair schedule are indicated in figure 4-2. Note that the two-job case along the left side of the graph shows beautiful

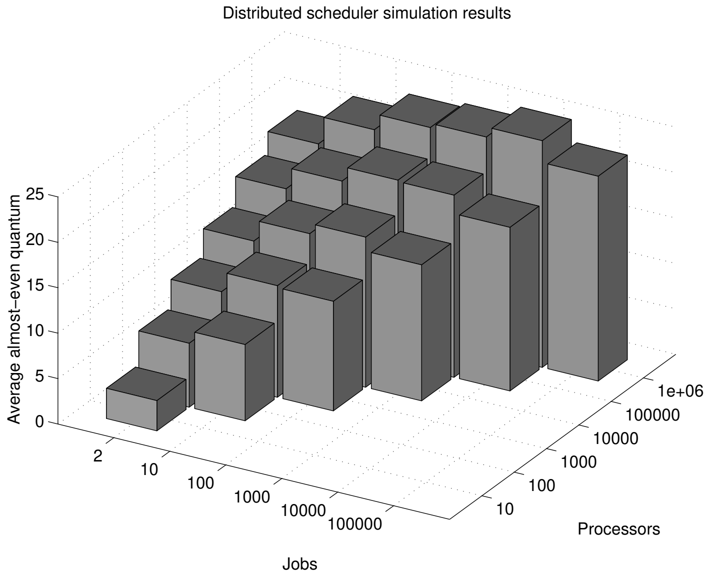


Figure 4-2: Distributed scheduler simulation results.

logarithmic growth, as does the $P = 10J$ case along the diagonal front. Some of the intermediate cases are a little higher than these end points, but it nonetheless appears that the algorithm would fit under a plane specified by $O(\log P + \log J)$, with the $P = 10J$ case doing better than this asymptotic bound. Since $P > J$ by assumption, $O(\log P + \log J) = O(\log P)$, which was the desired bound.

Chapter 5

Idleness

To utilize workstations only when they are "idle" or "unused" requires a concrete specification of when a workstation is idle, which may vary from installation to installation. Several indications, such as idle times and load averages, lend themselves to determining workstation idleness. These indicators, which have been used successfully in systems such as Piranha [5], must be combined with conditions on when they are applicable to form a useful and flexible set of idleness criteria. It is also desirable for the idleness criteria to be fairly non-volatile, so a machine declared idle tends to remain idle for at least a few minutes, preferably much longer, preventing the microscheduler from being forced to spend too much time adapting to the changing set of workstations. In this chapter, I discuss techniques for detecting idleness, define the macroscheduler's notion of idleness "predicates," and finally, I present experimental evidence that networks of workstations managed in this way present a useful computational resource.

5.1 Detecting idleness

This section discusses how idle times, the number of users, and load averages each provide some information about the idleness of a workstation [5]. Used in combination, they can gather large amounts of processing time which is practically transparent to the users of the workstations. Section 5.3 provides experimental results confirming this claim.

5.1.1 Idle Times

One useful measure of idleness is the time since the various input channels to a computer were last used. The macroscheduler can find this information both for the `tty` devices, which represent both local and remote users, and for the `keyboard` device, which represents a local user. Unfortunately, activity on some X windows which are displayed on a remote host do not have easily measurable idle times; in fact, kernel modifications are often required to detect such activity. For simplicity, the macroscheduler relies on other measures, mainly load averages, to try to detect this "invisible" activity.

5.1.2 Number of Users

Measuring the number of users on a system, i.e., the number of open `tty` ports, can provide some indication of the use of a system. Unfortunately, it is common practice for remote users to stay logged in to a machine with no activity for days, placing essentially no load on the machine. On the other hand, a user might be running a very processor-intensive application in the background without even being logged in. For these reasons, the number of users is not generally a very good gauge of workstation idleness, but is provided in the macroscheduler for the rare circumstances when this measure is meaningful.

5.1.3 Load Averages

Unlike idle times and number of users, load averages provide useful insight into whether a program has been left running unattended, as such a job does not show up in any of the user idle time measurements, but it increases the load average of a workstation. Load averages are a little tricky, though, since the parallel job being run increases the load average. Without special treatment, the parallel job would run until the load average became too high, and would then be killed. Then, the load average would fall, causing a parallel job to be started again. Fortunately, the algorithm used by most operating systems to compute load averages is a fairly straightforward weighted running average, so computing the effect of the single parallel job on the load average is simple. By subtracting these predicted effects from the load average before comparing with a threshold, the macroscheduler considers only the effect of the non-macroscheduled processes, i.e., the local users' jobs, when deciding on the idleness of the workstation.

5.2 Predicates

Some users may have particular demands for the workstations that they use, or for the workstation that sits on their desk: They don't want other people's jobs interfering with their work. For this reason, users can establish idleness conditions that only apply on their workstations or on whatever workstations they happen to be using. The system administrator can also establish system-wide global conditions, which serve as a minimum requirement for idleness. Each of these conditions is called a *predicate*. A predicate specifies one of the above types of conditions, i.e., idle time, number of users, or load average, along with associated parameters (e.g., `idletime=10` seconds), and a specification for when the predicate is applicable, i.e., always, on a particular workstation, when a certain user is logged in anywhere, or when a certain user is logged in on a particular workstation. In order for a workstations to be declared idle, all applicable predicates must be satisfied.

These predicates are maintained by the job broker and node managers, and can be viewed or updated using the `CilkPred` utility.

5.3 Experimental results

In this section, I present to sets of experimental results: The first shows that networks of workstations can provide a sizeable and useful computational resource, while the second supports the claim that a combination of idleness detection methods is required.

To evaluate the amount of time that these idleness criteria can gather, a 10-day experiment was run on the Theory of Computation group's network at M.I.T.'s Laboratory for Computer Science, consisting of 45 Sun Sparcstations and using two simple idleness criteria: an idle time of 15 minutes and load averages not to exceed 0.80, 0.60, and 0.50[1] (averaged over 1 minute, 5 minutes, and 15 minutes, respectively). Over this period, an average of 23.4 machines, or 52%, were working on the computation at any time. Figure 5-1 shows the the daily cycles in machine availability. Note that user activity differs by about a factor of two between nighttime (the idleness peaks) and daytime (the idleness dips). The average consecutive time a typical node participated was just under 40 minutes, though a couple of workers stayed for days at a time. The amount of processor time associated with different lengths of participation is shown in figure 5-2. These results show fairly high utilization

[1] These numbers are a little inflated since the load average check used here requires running a program each time the load average was to be checked. This is quite inefficient, and raises the load average a bit. The actual numbers used to consider a machine newly idle were 0.35, 0.30, and 0.25. This experiment was run using an oversimplified model of load averages, so slightly less time was gathered than would be by the final macroscheduler.

of the workstations for the computation, especially at night when the workstations were not being used much, with rather long blocks (typically 40 minutes) of continuous computation time.

A similar experiment run on 7 machines for about 20 days revealed that both of these idleness conditions were necessary. Idle time measurements but not load averages detected about 15% of the time workstations were used, while load average measurements[2] but not idle times detected 71% of this time. Both criteria concurrently detected only 14% of this time. Hence, each criterion alone would have missed a fairly considerable amount of user activity on these machines.

[2]Here the 0.35, 0.30, 0.25 numbers were used since the checks were performed less frequently and no jobs were being run during the experiment.

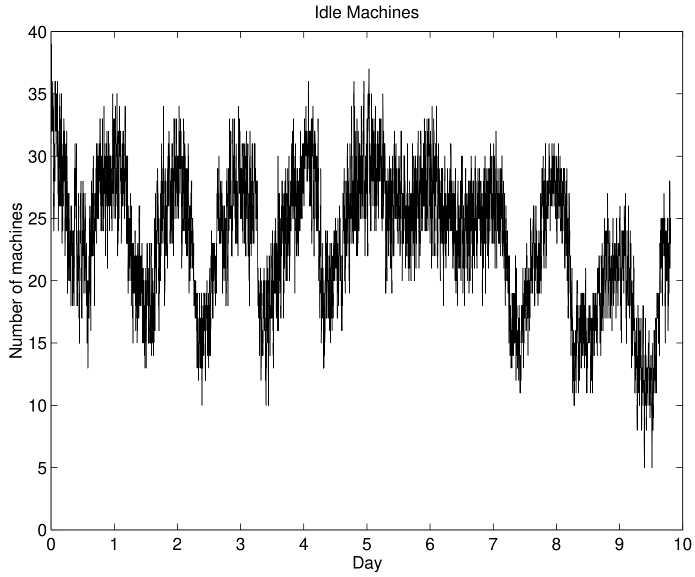


Figure 5-1: Participation in 10-day experiment.

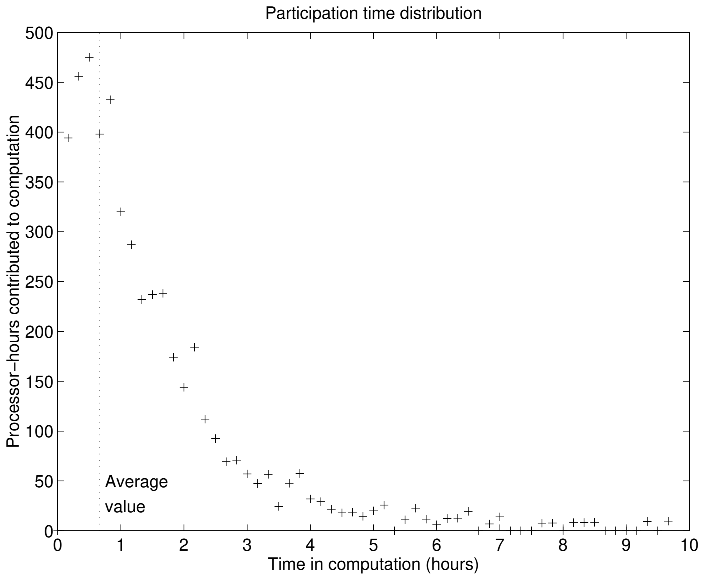


Figure 5-2: Participation time in 10-day experiment.

Chapter 6

Evaluation

The macroscheduler presented here succeeds in achieving its primary goals. It is easy to use, fault tolerant, flexible, and secure. Furthermore, the scheduler efficiently allocates workstations to various jobs using a simple distributed algorithm in $O(\log P)$ expected time, according to the simulations that have been run. Several ideas relevant to this thesis, such as refining fairness to account for parallel jobs that only need a few processors and a proof of the algorithm's convergence rate, however, warrant further exploration, and will be subsequently discussed.

The idea of fairness should include the notion that a job may not be nable to utilize its full share of processors at some point in time; when a job needs fewer processors than P/J, the excess processors should be distributed among those jobs that can utilize them. To facilitate this, the Cilk-NOW microscheduler, or other parallel job, must be able to indicate to the macroscheduler something about its current degree of parallelism. Monitoring the parallelism in a Cilk-NOW job might be accomplished by tracking steal attempts, whose rate is related to total communication, which in turn is closely related to the amount of work and the critical path length the job [3]. The steal rate might be used to estimate the average parallelism of a job. This information must then be integrated with the scheduling algorithm in such a way that jobs can grow rapidly when they need more processors, but not grow when the current supply of processors suffices. Likely, the microscheduler will take some amount of time, possibly a half a minute or more, to initialize itself and gather enough information about steal attempts to provide a reasonable idea as to the current degree of parallelism present. It would be preferable to have a scheduling algorithm that did not require waiting for these long amounts of time to perform a single quantum of the scheduling algorithm, as long quanta would slow down the scheduler significantly.

A proof of the convergence rate of the actual macroscheduling algorithm (or an approximated version of it) would also be beneficial. The derivation in chapter 4 suggests that this algorithm should be $O(\log P)$, and the the simulations concur, however a formal proof of this would be very helpful. Furthermore, a proof that when $J > 2$ the algorithm is still fairly efficient is needed. Currently, work is under way to provide such proofs.

Bibliography

[1] Robert Blumofe. *Managing Storage for Multithreaded Computations.* PhD dissertation, Massachusetts Institute of Technology, 1995.

[2] Robert Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing*, San Francisco, California, August 1994.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principal and Practice of Parallel Programming.*, Santa Barbara, California, July 1995.

[4] Eric A. Brewer and Robert Blumofe. *Strata: A multi-layer communications library.* M.I.T. Laboratory for Computer Science, February 1994.

[5] David Gelernter and David Kaminsky. Supercomputer with recycled garbage: Preliminary experience with Piranha. Technical report, Yale University, December 1991.

[6] Jospeh A. Kaplan and Michael I. Nelson. A comparison of queueing, cluster and distributed computing systems. Technical report, National Air and Space Administration, June 1994.

[7] Steven P. Miller, B. Clifford Neuman, Jeffrey I. Schiller, and Jermoe H. Saltzer. Kerberos authentication and authorization system. Athena technical plan, M.I.T. Project Athena, October 1988.

[8] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall Software Series. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[9] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1992.

[10] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.