Chapter 1

The ParPar System: A Software MPP

DROR G. FEITELSON, ANAT BATAT, GABRIEL BENHANOKH, DAVID ER-EL, YOAV ETSION, AVI KAVAS, TOMER KLAINER, URI LUBLIN, AND MARC A. VOLOVIC

Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
feit@cs.huji.ac.il

1.1 Introduction

To place ParPar[1]. in context, we must first review the different modes of operation common on clusters. Probably the most common approach is to view the cluster as a Network Of Workstations (NOW). With this approach, each node is owned by a certain individual, and is usually also physically located in his work area. The owner uses his workstation for administrative work, such as e-mail, and also for processing, e.g. text processing or engineering applications. But such work typically consumes only a fraction of the workstation's resources. The remaining resources are therefore available for general use by others, who need more resources than their local workstations can provide. Examples of projects based on this approach are the Berkeley NOW, Condor, and MOSIX [3].

At the other extreme are clusters dedicated to a single user at a time, with the explicit goal of executing parallel programs. The motivation is to provide resources for the solution of very demanding problems, and use them as efficiently as possible. Clusters, due to their use of commodity components, are cheap enough to make this possible. The best know system of this type is the Beowulf project. ParPar is related to this approach, but attempts to provide an inexpensive approximation of

[1] "Parpar" means "butterfly" in Hebrew, which is irrelevant; the name was chosen for its sound, because "par" is short for "parallel", and because two is better than one.

a complete MPP system, capable of servicing a general purpose multiuser workload of parallel programs. This enables the creation of a workload mix where different jobs complement each other, and together make better use of the resources.

While clusters usually emphasize the use of off-the-shelf technology, some do in fact use special components, especially for the network interface. For example, PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) implements barrier synchronization in hardware, and is competitive with MPP systems [6]. The price is, of course, added cost for both the components and their design, and more important, the need to support them through successive generations of the commodity components. We do not use any such special hardware.

In summary, the ParPar project aims to create a general-purpose, multiuser, MPP-like system, using only off-the-shelf components. The emphasis is on operating system functionality, such as job control and parallel I/O, as opposed to communication which has been handled by multiple other projects (e.g. Active Messages [16], Fast Messages [13], or U-Net [15]). The environment is created entirely in software. The next section presents an overview of the system structure; subsequent sections describe various system functions in some detail.

1.2 The ParPar System

In this section we describe the components of the ParPar system and their interrelations.

1.2.1 Hardware Base

The ParPar prototype cluster comprises 17 high-performance PCs: a host, called "parpar", and 16 nodes, called "par1" through "par16" (Fig. 1.1). The PCs have an Intel Providence motherboard with a Pentium Pro 200 processor, 128 MB DRAM with parity checking, and a 2.1 GB SCSI disk. Any other configuration could be used instead.

The nodes are connected by two independent networks. One is a switched Ethernet, that serves as the control network. All the system-oriented communication required to implement the functions described below passes on this network, using a combination of conventional TCP/IP and a reliable multicast protocol that we implemented. The other is a 1.28 Gb/s Myrinet [4], which is dedicated to the communication needs of user applications (this is called the data network). Communication is done using MPI over FM [13]. As user processes do not run on the host, the data network only connects the 16 nodes.

Future plans call for the inclusion of 4 I/O nodes in the system, that will also be connected to both networks. These nodes will also be high-performance PCs, and will have several larger disks. They will run a parallel file system that will support parallel access to files that are spread across all their disks (see Section 1.6.2). The currently available disks are only used for swap space and a local /tmp file system.

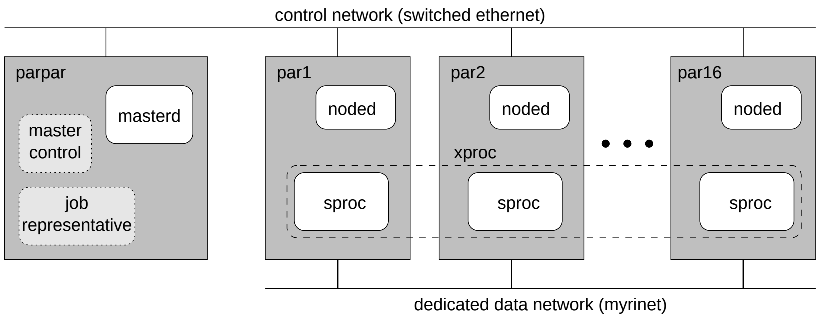

Figure 1.1. *Hardware components of the ParPar system, and software components that run on each one.*

1.2.2 Software Structure

The ParPar software is based on the Unix BSDI system, and runs at user level. The software components that make up the ParPar system are shown in Fig. 1.1. These include daemons that run on the host and on the nodes, and graphical user interfaces that run either on the host or on each user's workstation. In addition there are the user processes that make up the parallel applications executed by the system. The interactions among the different components are summarized in Fig. 1.2.

The master daemon (masterd) is the heart of the ParPar system. It is responsible for tracking system configuration (which nodes are up and part of the system) and for job management and resource allocation. There is one masterd in the system, and it runs on the host.

The node daemons (noded) provide local control over the processes running on the system's nodes. As such, they are agents acting for masterd. There is a single noded on each node.

The job representative (JR) is a GUI that provides an interface to the system and terminal I/O facilities (see Fig. 1.9). Users use the JR to launch jobs, by specifying the executable, its arguments, etc. The system displays status messages on the JR. Normally, there is a separate JR for each job running on the system. The JR can run on the host or on the user's workstation. A non-graphical job representative is also available for use in batch scripts.

The master control (MC) is a GUI that provides an interface for obtaining configuration and load information, and for controlling the system (e.g. shutdown). Normally, each user may invoke one MC interface, but this is not necessary in order to use the system.

Parallel jobs are known as extended processes (xproc), but this abstraction exists only on masterd and the JRs. Xprocs are composed of sub-processes (sproc) running on nodes. Sprocs connect directly to the job's JR to support terminal I/O activity.

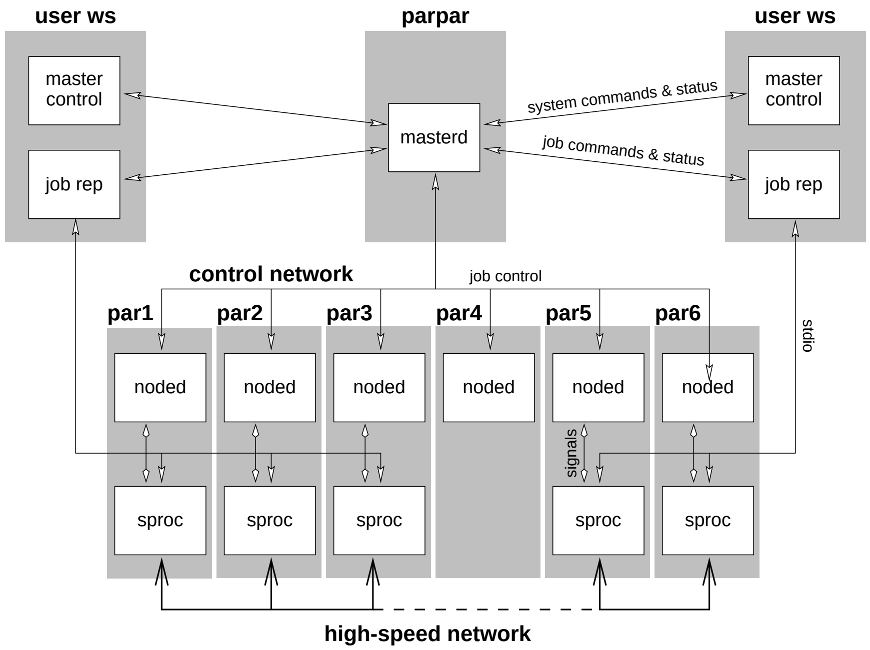

Figure 1.2. *Interactions among the system components.*

1.2.3 Design Principles

The system's software components described above are all reactive: most of the time they just wait for some input, be it user input to one of the GUIs or an incoming message to one of the daemons. The behavior and interactions among the different components are governed by the following two design principles, which are actually two facets of the same idea:

1. *Operations are atomic.* This means that when a system component receives some input, it reacts, and then returns to the wait-for-input state. Each message stands for itself. The daemons don't have any states in which they wait for a specific follow-up message from some other system component.

2. *Synchronism is minimized.* The system components, especially the daemons, operate in an asynchronous manner as much as possible. They do not block and wait for each other.

1.2.4 Control Protocols

The flow of control and information among the system components is outlined in Fig. 1.2. Obviously, the masterd is the hub of the system: all the GUIs and nodeds communicate with it. In addition, there are some direct communications from JRs

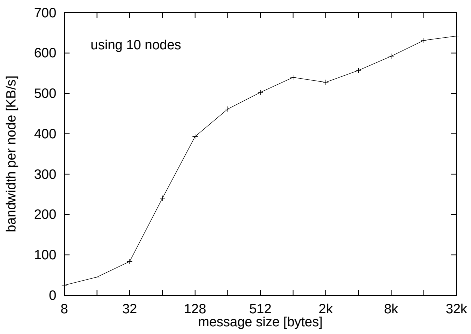

Figure 1.3. *Bandwidth achieved by the multicast protocol.*

to sprocs.

However, the topology does not give the whole picture. Focusing on the most important interactions, those between the masterd and the nodeds, we find that the following scenarios are common:

- The masterd notifies a set of nodeds about a new job that they should spawn.
- The masterd sends a (meta)signal to a set of nodeds, who forward it to the sprocs on their respective nodes.
- Nodeds notify the masterd about termination of sprocs.

Generalizing these interactions, we see that they always follow the same pattern: the masterd multicasts the same message to a set of nodeds, whereas the nodeds respond individually.

The naive way to implement such a communication pattern is to establish a TCP/IP connection from the masterd to each noded. The multicasts are then implemented as a loop in which the masterd sends the message to all the relevant nodeds. However, this suffers from increasing overhead as the system size increases, and does not utilize the broadcast capability of the underlying Ethernet medium. We therefore implemented a reliable multicast protocol based on UDP/IP.

The resulting communication structure is asymmetric: the masterd sends messages to nodeds using reliable multicast over UDP, and the nodeds send acks and other messages to the masterd using TCP. The multicast is in fact a broadcast with a destination bitmap. Thus the communication layer on all nodes receives all messages, but passes them to the noded only if the bit for this node is set. This allows a single numbering scheme to be used for all multicast packets.

Reliability is achieved by a sliding window protocol. The masterd may send a number of messages without waiting for immediate acks; instead, the messages are stored until acks are received from all the nodes. An ack for package i indicates that all packets up to and including i have been received. A nack on packet j indicates that all packets from $i+1$ (one after the last ack'd packet) to $j-1$ are missing,

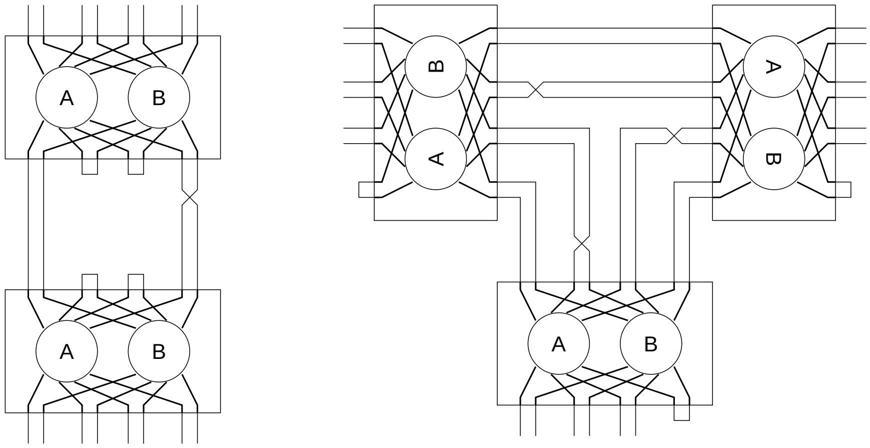

Figure 1.4. *Possible topologies for 16-node Myrinet networks (the 3-switch topology actually supports up to 18 nodes).*

but j itself was received. This allows masterd to retransmit only those packets that are indeed missing, instead of retransmitting everything after any loss of a single packet.

If neither an ack nor a nack arrive, the masterd re-transmits after a timeout period. The timeout period is calculated dynamically based on measurements of previous ack times, so as to avoid premature retransmissions. A weighted average of ack times is used, in order to adjust to changing network conditions. This is similar to the calculations used in TCP [5].

The bandwidht achieved by the multicast protocol is shown in Fig. 1.3. It peaks at just under 650 KB/s for 32 KB messages. This result is not sensitive to the number of nodes participating in the multicast.

1.2.5 Data Network

The data network provides user-level communication for application processes. This allows the crucial communication operations to circumvent the operating system, thus improving the bandwidth delivered to the application and reducing the latency. The interface used is the popular MPI standard. The implementation is a port of MPI-FM, the Illinois Fast Messages software [13]. Myrinet is currently the best commercially available LAN for this purpose [2].

A sufficient condition for Myrinet operation is that the network be connected, that is, that there be a path between any two nodes. However, such a topology may imply a much lower bandwidth between clusters of nodes connected to different Myrinet switches than within such clusters. In particular, the bisection bandwidth of the network may be as low as that of a single link. It is therefore advisable to use a richer topology.

Due to the specific switch configurations sold by Myricom, two dual switches are required in order to connect 16 nodes (a dual switch contains two eight-port switches referred to as A and B, but at least one port is needed to connect the switches to each other, so at most 14 nodes can be connected to a single dual switch). It is therefore possible to construct a topology with a bisection bandwidth of 4 links (Fig. 1.4 left). With three dual switches, a topology supporting 18 nodes with bisection bandwidth of 9 links is possible (Fig. 1.4 right).

1.3 System Configuration and Control

The previous section described the components of the full system. This section explains how the components recognize each other when the system is booted, and how they react to fault conditions.

1.3.1 Dynamic Reconfiguration

As far as the software daemons are concerned, the Configuration of the ParPar system is completely dynamic.

Upon startup, the masterd creates three sockets, binds them to well-known ports, and starts to listen on them. These ports are used for connections from nodeds, JRs, and MCs, respectively. In principle, the masterd does not have to know about these entities in advance in order to accept connections. In practice, we require the nodes to be listed in a configuration file, so that the full configuration can be loaded on each node when it connects. This is used to set up routing tables for the data network.

When a noded comes up, it enters a loop in which it tries to establish a connection with the masterd. As the noded is useless unless it is connected, it stays in the loop until a connection is made. Thereafter it serves as one of the nodes in the configuration maintained by the masterd. As part of the initial handshake, the node's serial number in the system is determined, and information about its capabilities and resources is registered with the masterd.

Node failures are identified by the masterd by the termination of the TCP connection with the noded. When this happens, the masterd updates its configuration information to reflect the fact that the node in question can no longer be used. This affects jobs running on this node, as described below, but does not affect the rest of the system.

1.3.2 Reliability and Availability

ParPar provides fault containment in the sense that the failure of any single node only affects jobs with sprocs running on that node, and even these jobs are not necessarily killed. In fact, node failure is simply regarded as the abnormal termination of the sprocs running on it. The details of handling abnormal termination are described below.

The masterd, on the other hand, is a single point of failure. This can be rectified by duplication or logging of all activities. However, we intend to perform measurements regarding the severity of this problem before complicating the system in order to solve it.

1.3.3 The Master Control

The MC GUI allows users to obtain information about the system configuration and load, and to perform certain operations on the system.

The basic items of information provided are a list of the connected nodes with their capabilities, and a list of the current jobs with their requirements and current status (e.g. running or queued). The items in each list are decorated with check-buttons, allowing selected nodes or jobs to be marked. Then an operation can be applied to the marked nodes or jobs.

For normal users, the only operations are to kill or stop their own jobs. System operators can also kill or stop jobs owned by others, shut down the whole system, or shut down a specific set of nodes. Stopping jobs is useful to enable performance measurements in a controlled environment. Shutdowns include a grace period in which jobs are allowed to terminate.

1.4 Job Control

A major objective of parallel systems is to run parallel jobs. This includes not only the initiation of jobs, but also control over jobs (e.g. being able to kill them) and debugging.

1.4.1 Job Initiation

Jobs are launched by users via the JR interface (Fig. 1.9). The minimal information required is the executable file name and the number of nodes to be used. It is also possible to provide arguments and specify what parts of the environment should be passed as well.

When the user hits the "load" button, the JR sends all this information to the masterd. The masterd is responsible for scheduling the new job, as described in Section 1.5. When the job is first scheduled, the masterd instructs the relevant nodeds to fork the required sprocs. Between the fork and the exec of the user's executable, the noded sets up the correct environment, redirects stdin, stdout, and stderr to sockets connected to the JR, and sets the user ID to that of the invoking user. Thus the job will have the correct permissions for accessing files.

Verification of the user ID claimed by the JR is based on a simple authentication protocol conducted when the JR first connects to the masterd. The masterd creates a file that is readable only by its owner, writes a random string into this file, and changes the owner to be the user claimed by the JR. It then challenges the JR to read the contents of the file. If the JR succeeds, the identity of the user is accepted.

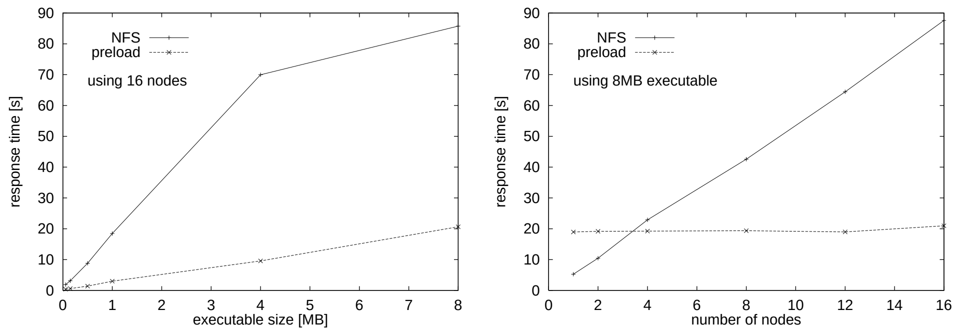

Figure 1.5. *Performance of executable file copying vs. NFS.*

The sproc can exec the user's executable in either of two ways. One is to use demand paging via NFS. The other is to first copy the executable to the local disk. In order to reduce the pressure on NFS, the copying is done through the masterd, which broadcasts the file to all the nodes. The performance implications are shown in Fig. 1.5. As the executable file size increases, the advantage of copying to the local disk becomes greater (left). Likewise, as more nodes are used, the advantage of local copying is increased (right). As the file is always copied to *all* the nodes, whereas NFS only brings pages to the nodes that are actually used, NFS is better for jobs that use only 1–3 nodes.

The masterd keeps a table of executables that have been copied. Thus if a user runs the same job again, the executable does not have to be copied again, but rather the existing copy is used. Verification that the executable has not changed is based on recording its last modification time. When the disk space to keep executables runs low, the ones that have been used the least number of times and the least recently are evicted. This is calculated as the sum of two terms that are both in the range [0, 1]. The first term is the fractional age of the executable: the time since it was last used, divided by the maximum over all executables of the time since the last use. The second term is the reciprocal of the number of times it was used. The executable with the highest score is evicted. Thus jobs that were run repeatedly are allowed to stay longer, out of anticipation that they will be used again.

1.4.2 Job Termination

Upon termination of an sproc, the noded that had forked it receives a SIGCHLD signal. It then uses the wait4 system call to obtain information about the cause of termination and the resources consumed.

If the sproc terminated normally, the noded simply forwards the information to the masterd, using a message we call a metaSIGCHLD. The masterd collects the information about all the sprocs, and when all of them terminate, it declares the whole xproc terminated and informs the JR.

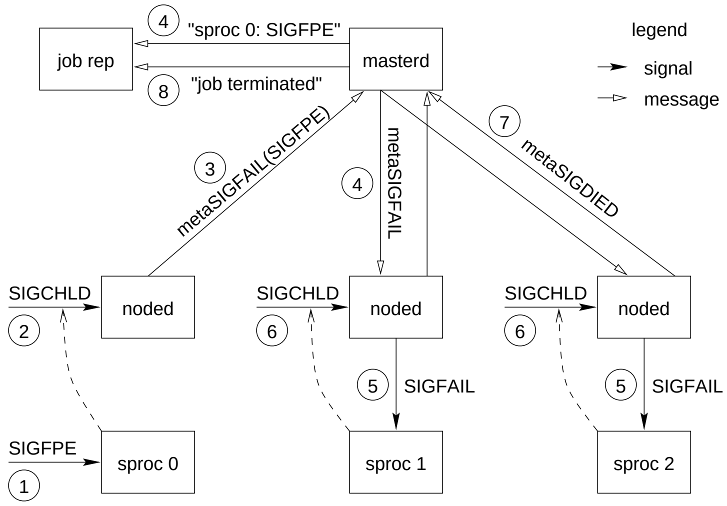

Figure 1.6. *Handling abnormal termination of sprocs.*

If the sproc terminated abnormally, the noded forwards the termination information to the masterd in the form of a metaSIGFAIL (an example is given in Fig. 1.6, where sproc 0 has a floating point exception). The masterd forwards this message to other nodeds that have sprocs of the same job. The nodeds then send a Unix SIGFAIL (implemented by SIGUSR1) to these sprocs. If the application was so coded, the sprocs can catch this signal and regroup to continue the computation. If not, the sprocs are killed, and the nodeds are notified by a SIGCHLD. The termination information is then forwarded to the masterd in the form of a metaSIGDIED, indicating that this is an expected abnormal termination, and need not be forwarded to other nodes.

Signals can also be sent by the user to all sprocs via and JR interface. In particular, SIGKILL is sent when the "kill" button is pressed. Such signals are sent in the form of a metaSIGSIG message from the JR to the masterd, which forwards them to the relevant nodeds, which then send the requested Unix signal to the sprocs.

1.4.3 Debugging

Debugging parallel jobs is difficult due to their concurrent nature. A debugger for parallel jobs must provide control over all the sprocs, and also be able to present concise status information.

The control structure used by the ParPar debugger is shown in Fig. 1.7. The debugger's graphical front-end is created by the JR when the job is loaded. Each noded forks a debugger back-end to control its local sproc. The back ends establish connections with the front end, and control the sprocs per the instructions they

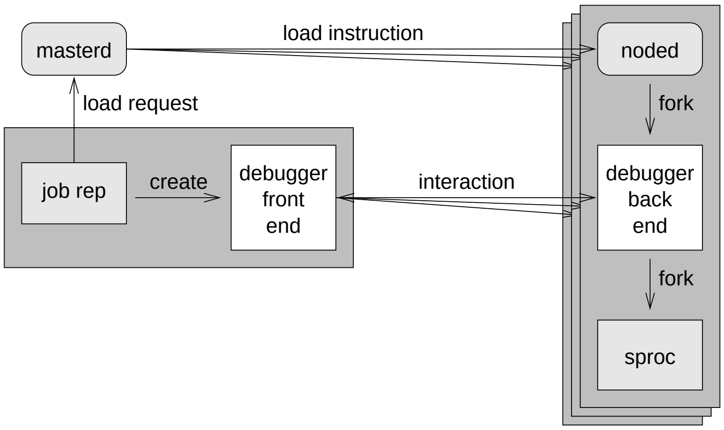

Figure 1.7. *Debugger control structure.*

receive.

A key feature of the front end is the notion of "focus". The focus is the set of sprocs being targeted at a given time. Debugger commands issued by the user, such as setting a breakpoint or stepping through the execution, are sent only to the sprocs in this set. Likewise, only information relating to these sprocs is displayed to the user. This mechanism allows the user to control single sprocs, the whole job, or any subset, using the same interface.

1.5 Scheduling

If the combined requirements of submitted jobs exceed the system resources, scheduling decisions have to be made. Two schemes have been implemented in ParPar: one based on space slicing, and the other adding a dimension of time slicing.

1.5.1 Adaptive Partitioning

Adaptive partitioning gives each job a dedicated partition of the machine, where the partition size is a compromise between the scheduler and the job that takes both the job's requirements and the current load into account.

Users can specify a set of possible partition sizes when a job is submitted, rather than demanding a single size. For example, in Fig. 1.9, the sizes 2, 4, 6, or 8 have been requested. The scheduler then allocates the largest size that is available. This has the desired outcome: the system automatically allocates the largest size that is available that is suitable for the job, thus allowing it to start running as soon as possible.

If none of the desired sizes is available, the job is queued and awaits the termination of some running jobs. When more processors become available, jobs are

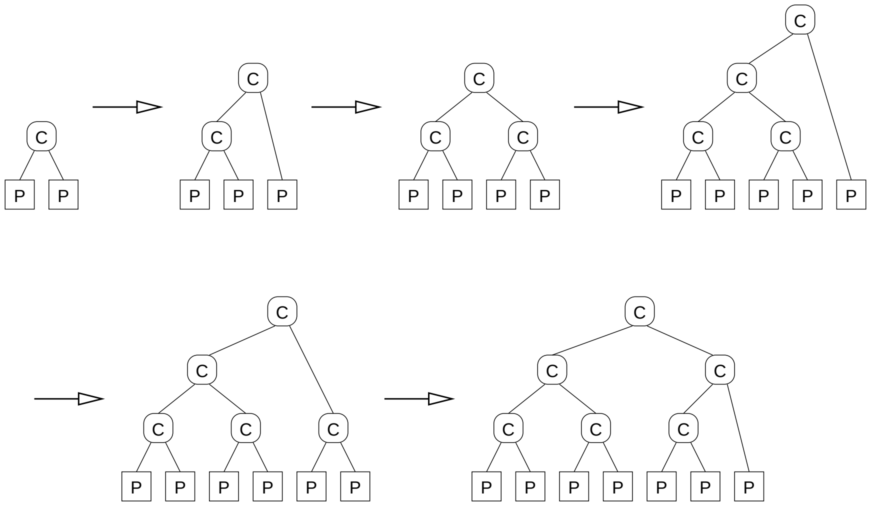

Figure 1.8. *Growing the Distributed Hierarchical Control structure incrementally from 2 to 7 nodes.*

started in FIFO order from the queue. But if the first job in the queue requires more nodes than are available, subsequent smaller jobs may skip over it. This is limited to a predefined number of times, e.g. 5, so as not to starve the large jobs.

1.5.2 Gang Scheduling

Gang scheduling provides coordinated time slicing among multiple jobs. At any given time, either all the sprocs in a job are running on distinct processors, or none are running.

The first issue in the implementation of gang scheduling is how to map sprocs to nodes. We use an extension of the Distributed Hierarchical Control framework, which defines a binary tree of control points [11]. The extension supports non-power-of-two systems, and builds the control tree incrementally as additional nodeds join the system (Fig. 1.8). This scheme nevertheless tends to pack jobs using groups of processors that are powers of two, which improves the ability to perform alternative scheduling. This is crucial in order to achieve high utilization and reduce fragmentation [8].

The second issue is the implementation of the coordinated context switching across multiple nodes. This is done by metaSIGSWITCH messages that are broadcast from the masterd to the nodeds. Upon receipt of such a message, the nodeds send a SIGSTOP to the currently running sproc on their node, and a SIGCONT to the designated sproc that should run next. As all the nodeds do this at about the same time, the effect is that of descheduling the current xproc and scheduling another in its place.

A potential problem with gang scheduling is memory pressure, because multiple jobs have to be memory resident at the same time. If the sum of the working sets of the sprocs mapped to the same node exceeds the available memory, thrashing will ensue. We have therefore implemented a version of the gang scheduler that includes such memory considerations. This version is able to queue or swap additional jobs rather than running them all at the same time, so as to reduce the danger of excessive paging.

A major problem in dealing with memory requirements is to assess how much memory is needed. Our scheduler bases this assessment on two inputs: an analysis of the executable file, and historical data from previous runs of the same executable. While not perfect, this provides estimates that are within a few percentage points of actual usage more than 90% of the time.

Given an estimate of how much memory is needed, the scheduler searches for the least-loaded set of processors that can run the submitted job. The load function is the sum of two terms: the number of jobs already scheduled on this set of processors, and the memory load function. If enough free memory is available, the memory load is 0. If it is exhausted, the memory load increases so as to represent the extra delays expected due to paging.

1.6 Parallel I/O

I/O falls into two categories: terminal I/O and file I/O. Both have special requirements and characteristics in a parallel system.

1.6.1 Terminal I/O

Terminal I/O is a much neglected issue in many parallel systems. However, it is a useful device, especially during program development. ParPar includes extensive support for efficient control and use of terminal I/O.

The most common form of terminal I/O is text: the user types text at the keyboard, and views text on the screen. In a parallel system, this is complicated by the fact that a single keyboard and screen must connect to multiple sprocs running on distinct nodes. It is therefore necessary to be able to control exactly which sprocs should be involved. In ParPar, this is done by input and output menus, that are located just above the standard I/O window in the JR (Fig. 1.9). By default, all nodes are included. However, any subset may be selected instead. Input typed at the keyboard is only sent to nodes indicated in the input set. Output from the sprocs is only displayed if they are in the output set. In addition, the node's serial number is indicated at the beginning of each line of output.

An innovative form of output supported on ParPar is a so called LED array [9]. Part of the JR's panel is partitioned into a set of small squares. The number of such squares per node is setable by the application. The application can also color these squares in different colors, causing an effect similar to flashing LEDs on the front panel of a machine. This is convenient for displaying the status of the program.

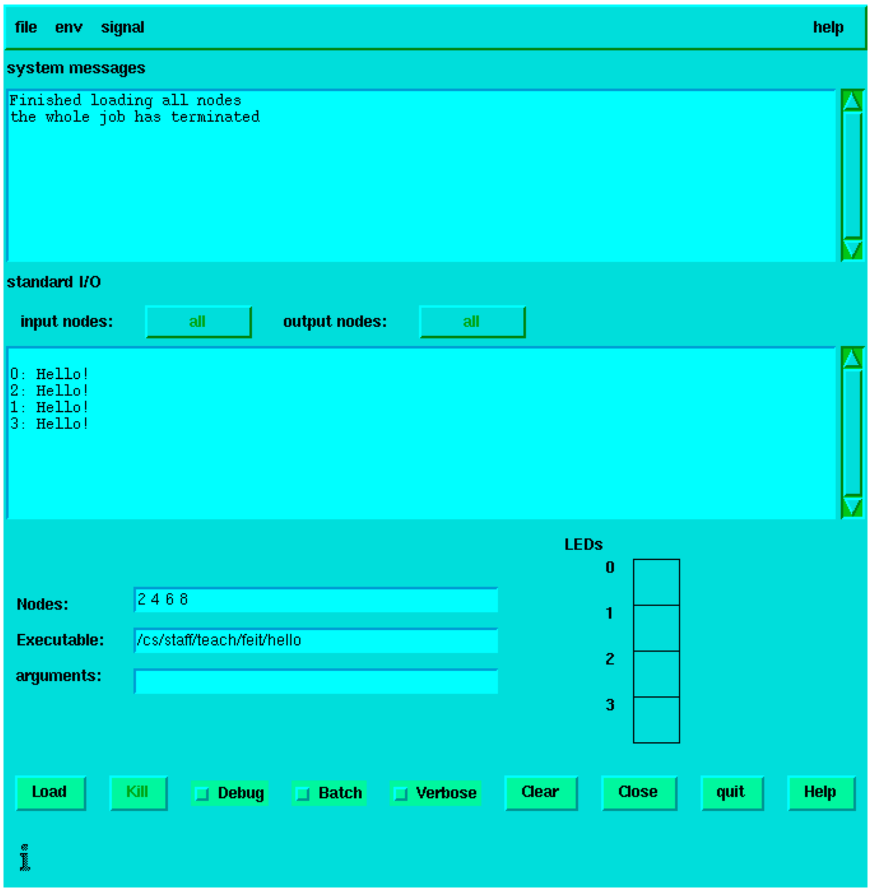

Figure 1.9. *The Job Representative GUI, after running a "hellow world" program on 4 nodes.*

For example, instead of printing a string to the terminal from each node to indicate that "initialization complete; starting computation", one can change the color of a LED from red to blue. The user sitting at the terminal will immediately see if all nodes perform the transition correctly at the beginning of the execution, or some remain stuck in the initialization phase. This is easier than having to locate and count the printed outputs which may be interleaved with other printouts and with inputs.

1.6.2 Parallel Files

The initial prototype of the ParPar system circumvents the issue of parallel file I/O by relying on two mechanisms. First, user files are accessed using NFS services. As each node runs a full Unix system, it is simple to mount the user's file systems on all the nodes. It is up to the application programmer to ensure that different processes don't interfere with each other, for example by writing to the same file. The second is a large /tmp space (about 1GB) that is provided locally on each node. This is useful for the implementation of out-of-core algorithms.

While NFS provides convenient support for remote file access, it has many drawbacks when used by a parallel program. In parallel programs, concurrent access to the same file by numerous processes is the rule rather than the exception. The NFS design assumes the opposite. As a result, various NFS design decisions complicate the use of NFS files by parallel programs. In particular, NFS's caching policy implies that if multiple processes write *disjoint* parts of the same file, they might still corrupt each other's data. Moreover, NFS might also become a performance bottleneck, as each file is ultimately handled by a single server.

The solution is to use a parallel file system within the parallel machine [10]. We plan to add 4 dedicated I/O nodes to ParPar for this purpose. These nodes will run a parallel file system that provides efficient support for file partitioning.

Studies of parallel file access patterns have revealed that in many cases processes make multi-dimensional strided access patterns [12]. For example, a process may access contiguous spans of 16 bytes that are separated by strides of 1024 bytes. These access patterns correspond to rectilinear partitions of multidimensional data structures, such as rows or blocks of a matrix. Typically, all the processes taken together access the whole data structure, but each one only accesses its partition.

Partitioned access, by its nature, involves multiple requests for small parts of the file. Performing such requests explicitly leads to inefficient use of the disks, which are optimized for large sequential access. It is therefore imperative to inform the file system about the global access pattern, so as to optimize disk access. The file system is then responsible for reorganizing the data and distributing it to the different processes as needed. In order to do so, the programmer must specify the access pattern in advance. Several interfaces for this purpose have been proposed, including the MPI-IO interface that is part of MPI-2.

We intend to integrate the idea of file partitioning with the mechanisms of caching and prefetching in the file system. Caching and prefetching are well-known techniques to improve performance, but they suffer from problems of coherence and false sharing when applied to parallel files. Our solution is to apply these mechanisms at the logical level of file partitions rather than at the physical level of disk blocks. In other words, blocks of logical partitions will be cached at compute nodes, rather than disk blocks (Fig. 1.10). This will have profound implications: it will simplify the implementation of these mechanisms, will improve their performance due to the increased possibility of caching and the reduced communication, and might even eliminate the need for additional more complicated and costly performance optimizations.

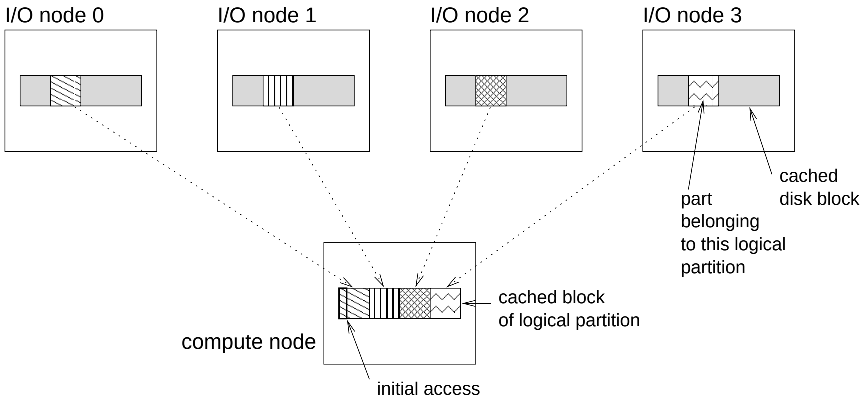

Figure 1.10. *Caching logical partitions on compute nodes.*

1.7 Project Status

At the time of writing (October 1998), a cluster of 17 Pentium-Pro 200 machines connected by both Ethernet and Myrinet networks has been acquired, and a first prototype of the system is essentially operational. Of all the features described above, only a the debugger and MC GUIs and the parallel file system are not finished yet. This cluster is being used for two purposes: first, we are conducting research on additional enhancements to the system software. Initial examples of such research are the use of reliable multicast for control functions, and memory-cognizant gang scheduling, as described above. Ongoing and future projects include logical caching in the parallel file system, and improved integration of the user-level communication library. Second, we are nurturing a budding collaboration with users from the Physics department who intend to use the system to run large-scale numerical computations. This will both help in exposing the strengths and weaknesses of the system design, and allow us to observe real usage patterns. Such observations will then be used to guide future developments.

For updates and the ParPar design document, see URL

http://www.cs.huji.ac.il/labs/parallel/parpar.html.

Acknowledgements

This research was supported in part by The Israel Science Foundation founded by the Israel Academy of Sciences & Humanities, and by the Ministry of Science.

1.8 Bibliography

[1] H. Bal, R. Hofman, and K. Verstoep, "*A comparison of three high speed networks for parallel cluster computing*". In *Communication and Archi-*

tectural Support for Network-Based Parallel Computing, D. K. Panda and C. B. Stunkel (eds.), pp. 184–197, Springer-Verlag, Feb 1997. Lect. Notes Comput. Sci. vol. 1199.

[2] A. Barak, S. Guday, and R. G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag, 1993. Lect. Notes Comput. Sci. vol. 672.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W-K. Su, "*Myrinet: a gigabit-per-second local area network*". *IEEE Micro* **15(1)**, pp. 29–36, Feb 1995.

[4] D. E. Comer, *Internetworking with TCP/IP, Vol. I: Principles, Protocols, and Architecture*. Prentice-Hall, 3rd ed., 1995.

[5] H. G. Dietz, R. Hoare, and T. Mattox, "*A fine-grain parallel architecture based on barrier synchronization*". In *Intl. Conf. Parallel Processing*, vol. I, pp. 247–250, Aug 1996.

[6] D. G. Feitelson, "*Packing schemes for gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 89–110, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[7] D. G. Feitelson, "*Terminal I/O for massively parallel systems*". In *Scalable High-Performance Comput. Conf.*, pp. 263–270, May 1994.

[8] D. G. Feitelson, P. F. Corbett, S. J. Baylor, and Y. Hsu, "*Parallel I/O subsystems in massively parallel supercomputers*". *IEEE Parallel & Distributed Technology* **3(3)**, pp. 33–47, Fall 1995.

[9] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[10] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best, "*File-access characteristics of parallel scientific workloads*". *IEEE Trans. Parallel & Distributed Syst.* **7(10)**, pp. 1075–1089, Oct 1996.

[11] S. Pakin, V. Karamcheti, and A. A. Chien, "*Fast messages: efficient, portable communication for workstation clusters and MPPs*". *IEEE Concurrency* **5(2)**, pp. 60–73, Apr-Jun 1997.

[12] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "*U-Net: a user-level network interface for parallel and distributed computing*". In 15th *Symp. Operating Systems Principles*, pp. 40–53, Dec 1995.

[13] T. von Eiken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "*Active messages: a mechanism for integrated communication and computation*". In 19th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 256–266, May 1992.