ParPar Design Document
Version 0.2— Preliminary Draft

Dror G. Feitelson
Marc A. Volovic
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

June 3, 1997

Contents

Part I
Background

1 Introduction

The ParPar system is intended to become the main research vehicle of the parallel systems lab at The Hebrew University's Institute of Computer Science. As such it replaces the older Makbilan system.

The Makbilan was based on sixteen i386-based Single-Board Computers, connected by a Multibus-II. Each board had 4MB of memory, and could access the memories of other boards via the bus, creating a NUMA shared memory architecture. While this machine was quite powerful when it was first constructed in 1989 (we even published some papers! [2, 6, 8, 11, 13, 22]), it has since fallen behind common desktop machines.

ParPar is meant to provide us with an updated research platform. It will be a distributed memory machine, in which Pentium-based PCs are connected by a high-speed local network. Using off-the-shelf PCs provides integrated power and simple packaging, reducing the construction effort. This also allows for individual I/O devices, thus opening the way for research in this area.

In a nutshell, ParPar is a software MPP. It is a combination of off-the-shelf hardware with our software, in order to emulate an MPP efficiently and unexpensively. It is not a NOW or COW [1]: rather, the system is dedicated to serving paralel jobs, at the possible expense of full Unix compatibility.

2 Project Goals

The foremost goal of this project is to provide a platform for research on parallel software, with an emphasis on system software. Such research often includes components of design, analysis, and simulation. With ParPar, we will be able to progress to real implementations and performance measurements as well.

To achieve this goal the following subgoals are required:

- The system should be a full system. This means that all aspects of a working system are covered, including program execution, debugging, I/O, etc. Initially all these features will be provided in the simplest manner possible. Later the prototype versions will be replaced by more sophisticated versions, partly based on our research on the different issues.

- The system should be convenient and easy to use. It should not require a lot of tutoring before users can make progress.

- The system should be flexible and modular, so that different versions of various components can be integrated together.
- The system should include facilities for measurement and monitoring that will be required for the research.
- Things that are given and cannot be modified (e.g. software we get from others without source code) should be minimized.

Note that these are goals for creating the infrastructure for research on parallel systems, not the goals of the research itself.

As research on parallel systems depends on measuring their performance, and measurement depends on the workload, a secondary goal is to open the system to real users who need to run parallel programs. This implies, again, that a full system implementation is needed, and also that conventional programming interfaces be used as much as possible.

3 Assumptions About Hardware Environment

The system will be based on off-the-shelf Pentium PCs. Each PC will include a sizeable memory (32 or 64MB), and a sizeable disk (1GB). A modest system size is expected, maybe up to a few dozen nodes at best. The design point specifies **a limit of 256 nodes.**

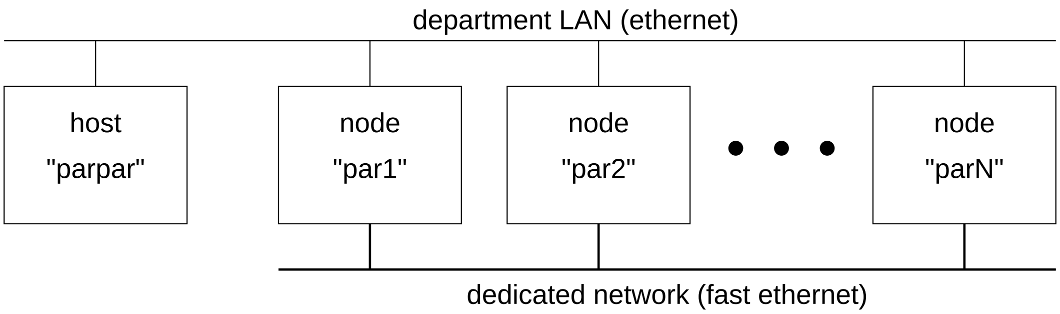


Figure 1: Sketch of system configuration.

The structure of the system is sketched in Fig. 1. This is a host-node design. The host, called *parpar*, is also a Pentium PC like the others, apart from having more disk space (and different software). It is used for system administration and resource management. It can also be used by users to launch jobs, but this function can also be performed from any other machine that is connected via the department LAN.

ParPar is connected to the node computers (called *par1* through *parN*) by a control LAN. This is simply the department LAN, which is an Ethernet. Later, we might use a dedicated segment, in order to further isolate the nodes from external influences, while maintaining connectivity when needed (e.g. for NFS mounting of users' home directories).

This network is used by the system software for control tasks such as coordinating job loading and termination, using sockets and the TCP/IP protocol suite.

The nodes themselves are also connected by a dedicated network, which initially will be a fast Ethernet, and later might be a myrinet [3]. This is specifically intended only for user level communication within parallel programs, although some usage by the system as well may be possible (e.g. to broadcast load modules or coordinate gang scheduling). It is desirable to have a programmable network adapter for the dedicated network. This will facilitate efficient implementation of user-level communication, and offload overhead from each node's processor. However, initially such an adapter is not assumed.

4 Functional Requirements

The obvious central functional requirement of the system is to be able to run parallel jobs. But how this is done will change as the project develops.

In the initial version, being able to run parallel jobs is the only major requirement. A parallel job is called an **xprocess**, for extended process. Each constituent of the xprocess is called as **sprocess**, for subprocess (this and other terminology is listed in section 35). Requests to run jobs will be submitted to the master daemon on parpar. The master daemon will allocate nodes to the jobs, load them, and start their execution. Nodes will be allocated exclusively to jobs for the duration of execution, i.e. space slicing. The goal of this version of the system is to erect an initial scaffolding of the control software structure, although some research on space slicing is also possible.

Running jobs is just one aspect of job control. In addition, a user must be able to kill jobs cleanly and without leaving any stray processes on the master or on the nodes. Of course, the resource manager should also stay consistent. In addition, facilities for controlled execution and debugging are planned.

The next version of the system, which is intended to be the base system for most future research, will include gang scheduling and user level communication. Gang scheduling means that time slicing is used in addition to space slicing, and moreover, that context switching is coordinated across the different nodes. User-level communication means that application programs have access to the communication network without operating system intervention. This also means that the system cannot guarantee security and therefore it should only be permitted when it is ensured that the processes running on other accessible processors belong to the same job. This is one of the reasons for using gang scheduling.

The communication API will be standard — both MPI and PVM will be supported, so as to attract potential users. These will be based on porting public domain code. It is anticipated that in the future some public domain DSM system will also be ported.

Finally, given that the system is intended for research on parallel systems, monitoring and measurement facilities are required. These will include access to the Pentium hardware counters on each node, as well as system-level information. In addition, all relevant system activity should be logged.

5 Project History and Plan

The project is planned to span a number of phases:

Phase I: Initial development. This version of the system will only provide the basic infrastructure to be able to run parallel jobs. The major software components (master daemon, node daemons, and job representatives) will be in place, but their functionality will be limited. In particular, space slicing will be used with no gang scheduling, and Unix communication will be used rather than user-level communication.

Phase II: Basic system. This phase will complete the basic system functionality, and will provide a research platform for further work. In particular, this phase will implement gang scheduling, user-level communication, and a basic parallel debugger.

Phase III: Research system. In this phase the system will be used for various research projects. Examples include:

- Research on memory management, paging, and swapping in parallel systems.
- Research on parallel I/O, parallel file systems, and file partitioning.
- Research on programming aids and debugging.
- Research on support for real-time computing that requires performance guarantees.

In addition, some more features will be added to the base system by means of porting existing software, such as

- Support for distributed shared memory.
- Program development and monitoring tools.

Preliminary design of the first phase was started in June, 1996, and has led to the production of version 0.1 this document. Implementation of the first phase was started in October 1996, and is expected to be completed in early summer 1997. The status at this stage is described in version 0.2.

Part II
Basic Concepts

This part of the document describes the basic concepts and general design of the ParPar system. This material is repeated in more detail in Parts III through VI.

6 Basic Software Design

We most emphatically do not want to develop the whole system from scratch. Therefore the software will be based on an available local kernel that runs on each node and supplies the required low-level functionality (context switching, memory management, device drivers, etc.).

The kernel we have chosen is the Unix BSDI system, for the following reasons:

- It is a supported commercial product.
- We have a site source license for it and can run it on any number of machines.
- It will be used by the next generation of servers in our department, so we can get help from the system group.
- It has a reputation for a good implementation of network services, which we need as the basis for system control.
- Unix in general is relatively well known in our environment (more so than NT or QNX).

While the choice of Unix is pretty certain at this point, the exact version is debatable. The main contender is Linux, which is also free with source [4]. Linux enjoys more use as a basis of research, and has more software available (e.g. weird and wonderous device drivers), but it is more volatile due to being maintained and modified concurrently by a wide and diverse community of users worldwide.

It should be noted that using Unix also has some drawbacks. The main one is that a full Unix system on each node is more than we need. Therefore we will have to strip down the system, e.g. by unconfiguring certain subsystems and removing various daemons. For example, we do not want the daemons that service various remote operations such as finger, rsh, etc. The list of what will be deleted is given in Section 10.3.

The basic software configuration is sketched in Fig. 2, and includes the following components:

- The master daemon, running on the host system (parpar), is the heart of the system. It is responsible for configuration management (i.e. keeping track of what nodes are in the system) and resource management (allocation of nodes to jobs).

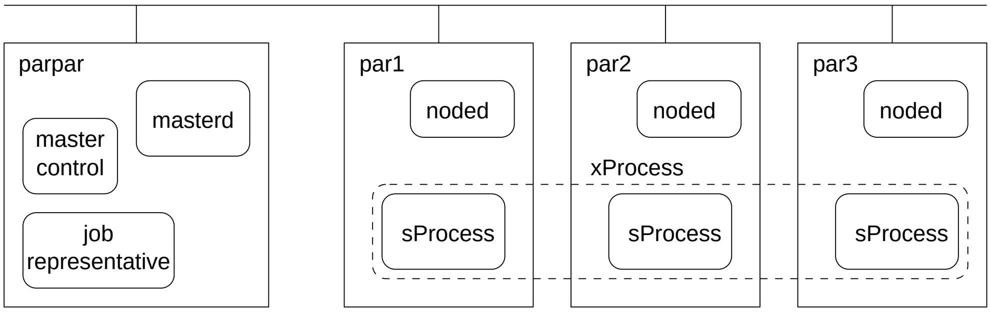


Figure 2: Sketch of software configuration.

- The master control is a graphical user interface that allows users to query the master daemon about system status. The super user can also use this interface for actual system control, e.g. deleting nodes or shutting down the system. It can run on any workstation, communicating with masterd via sockets.
- The job representative is a graphical user interface used to control jobs. There is one job representative for each job, running on the user's workstation or the host (wherever the user logged in).
- The node daemons are local daemons that execute on each node. Their main function is to fork the sprocesses that constitute parallel jobs, and forward signals to them. They are also responsible for coordinating the gang scheduling.
- Sprocesses are user processes that together form a parallel job (called an xprocess, for "eXtended process"). They are linked with an application library that supplies functions for obtaining configuration information and message passing.

The ways in which these software components interact to perform various system functions are described below. The general philosophy is to keep things as independent and asynchronous as possible. The mechanics of communication are to use sockets with the TCP/IP protocol suite, over the control network.

In the future, certain functions (most notably support for gang scheduling) may migrate into the kernel.

7 Job Execution and Termination

7.1 Job Execution

The user is responsible for preparing a suitable load module, typically be compiling a C program and linking it with the appropriate ParPar libraries for communication etc. Initially the same load module will be loaded on all the nodes, leading to an SPMD style. In the future, support for MIMD (i.e. different load modules) is possible.

There are two options regarding how the user performs job control, and specifically how jobs are loaded and killed. One is to try to make parallel jobs look as much like normal Unix jobs as possible, and the other is to use a special tool for parallel job control.

In the Unix-like approach, the compilation generates a normal Unix executable (probably a shell script) that is executed by the user like any other program. This shell script interacts with the ParPar master daemon to allocate nodes and load the real executable on them. In addition, some mechanism is needed to notice if the user kills this front-end script, so as to kill the job as well. Finally, standard I/O should be piped from the processes on the nodes to the display used by the user, and vice versa.

In the special tool approach, a separate window is opened for interaction with the ParPar system. A sketch of a possible layout is described in detail in Section 19, and shown in Fig. 6. This window includes a sub-window for standard I/O, entry widgets for specifying the number of nodes to use, the load module, and any command line arguments, and buttons to initiate the load operation and to kill a running program.

As it seems that using a special tool is simpler and more versatile than trying to make parallel jobs look and behave like Unix jobs, we will use a special tool for job control. This is called the **job representative**. If a user wishes to run multiple jobs at the same time, a separate job representative is needed for each one.

There are also two options for the mechanics of starting up the sprocesses. The job representative can directly use rsh to start an sprocess on each allocated node, or the nodeds can fork/exec the sprocesses locally based on information provided via messages. The first approach is simpler because Unix takes care of the details like user ID, file access, etc. However, we chose the second approach because it is expected to be more efficient [23], gives more control (so there is a lesser chance of stray processes remaining in the system), and is more amenable for incorporating gang scheduling.

Once a job is running, file I/O is supported via conventional NFS mounting on the nodes. In the future, a parallel file system will be developed as part of the research effort.

7.2 Job Termination

Jobs may terminate in one of three ways:

1. normal termination at completion,
2. abnormal termination due to an internal exception, e.g. divide by zero,
3. killed by external agent, typically the user.

In the first case the daemon on each node should notify the master daemon that the sprocess has terminated. The master daemon will accumulate this information, and when all the sprocesses terminate will announce that the whole job has terminated.

In the second case, the local daemon of the faulting sprocess should again notify the master daemon. The master daemon will forward the information to all the other node daemons. This mechanism is called a **meta-signal**. The node daemons, in turn, will send a signal to their local sprocesses (we call this a SIGFAIL). If the application is written in a fault tolerant mode, this signal will be caught and the remaining sprocesses will reorganize to complete the computation. Otherwise they will terminate, and a suitable message should be sent to the job representative for display to the user.

In the third case, the initiative comes from the job representative or master daemon. A notification is sent to all the relevant node daemons to kill the participating sprocesses, using the KILL signal.

7.3 Debugging

The above scenario is much complicated when debugging requirements are introduced. A major issue is controlled execution of multiple independent processes on distinct nodes, and an interface to express exactly what is desired. This will probably be an extension of a conventional debugger divided into two modules: a front end that will deal with multiple nodes by focusing on single nodes or subsets of nodes, and a back end that will control the individual sprocesses. A detailed description is given in Section 30.

The front end will have a graphical user interface, similar to that of the job representative, but with some changes. Included features will be a display of program text with markers of breakpoints and the current location of different sprocesses, a color-coded node status map, and an interaction window to input debugger commands and receive responses from the nodes.

8 Application Programming Environment

The system will provide various options for writing parallel applications, but will also impose certain restrictions. This section describes the main points. The basic guideline for the design is a quest for simplicity and standardization. Our research effort is in the direction of system software, not defining new interfaces. On the contrary, we wish to leverage known and accepted interfaces to enable users to use our system more easily.

The system will initially support an SPMD programming style. This means that a parallel job will be composed of multiple copies of the same load module executing simultaneously on different processors. In the future, support for a MIMD style (i.e. different load modules) is possible.

The environment provided for the individual processes is a combination of Unix and special additions for parallel programming. The Unix part is somewhat restricted. It includes most of the Unix support routines, such as I/O and timing. It specifically excludes most of the routines that handle processes, such as `fork`, `exec`, `kill`, etc. Initially, we will leave the outcome when such functions are called undefined. In the future, it is intended to catch these functions and send a signal to the application that called them.

The extensions for parallel programming will include conventional interfaces such as PVM [14, 24] and MPI [18, 7]. These interfaces provide support for message passing and for getting information about the system configuration (e.g. how many processors is the application using). However, these interfaces will not be supported in full. For example, PVM includes functions for dynamic resource changes (add or delete processes), which we will not support initially. A detailed list of supported and unsupported system and function calls is provided in section 12.

As for file I/O, initially this will be supported via NFS mounting on the nodes (it is assumed that the same file systems are mounted on the host and on all the nodes, by means of the departmental LAN) [21]. As this is an inadequate solution for high bandwidth parallel I/O, we intend to later study the possible implementation of an internal parallel file system within ParPar. Standard I/O is piped to the job representative, and includes support for stdin, stdout, and stderr. In addition, we include the notion of "LEDs" — small colorful squares that can be set under program control [10]. LEDs are explained more fully in Section 14.3.

9 Detailed Design Roadmap

The rest of this document presents a more detailed design of the concepts discussed above. It is organized into four parts, as follows:

Part III: basic functional procedures.
This part describes the basic control flow and procedures used to boot the machine, run parallel programs, and handle faults. Note that running parallel jobs includes not only starting them up, but also supporting them while they run and terminating them cleanly...

Part IV: software components.
This part describes how the machine works but from a different perspective, namely based on the software components. Each component is described, including its participation in each of the procedures discussed above, data structures it maintains, etc.

Part V: communication protocols.
This part describes the protocols between each pair of communicating software components, including detailed layout of messages exchanged.

Part VI: additional services.
This part describes additional services that were not included in the "basic procedures" part above, although they probably belong there. These services include support for logging, debugging, gang scheduling, and user-level communication.

Then comes an appendix that holds a list of open questions, a glossary of terminology as we use it, coding standards, and references.

Part III
Control Flow and Basic Procedures

The core of a parallel system in the ability to run parallel programs. The basic procedures required therefore include procedures for parallel program loading, support for parallel program execution, and parallel program termination. In addition, it is necessary to support parallel machine startup (booting) and fault handling. These building blocks are detailed in the sections below.

The software components of the ParPar system were shown in Fig. 2, but without their communication structure. Here we start to explain how the different components interact with each other. The description in the following sections is redundant in the sense that most things are mentioned twice: once in the sections about the pertinent software modules, and again in the section about the control protocol.

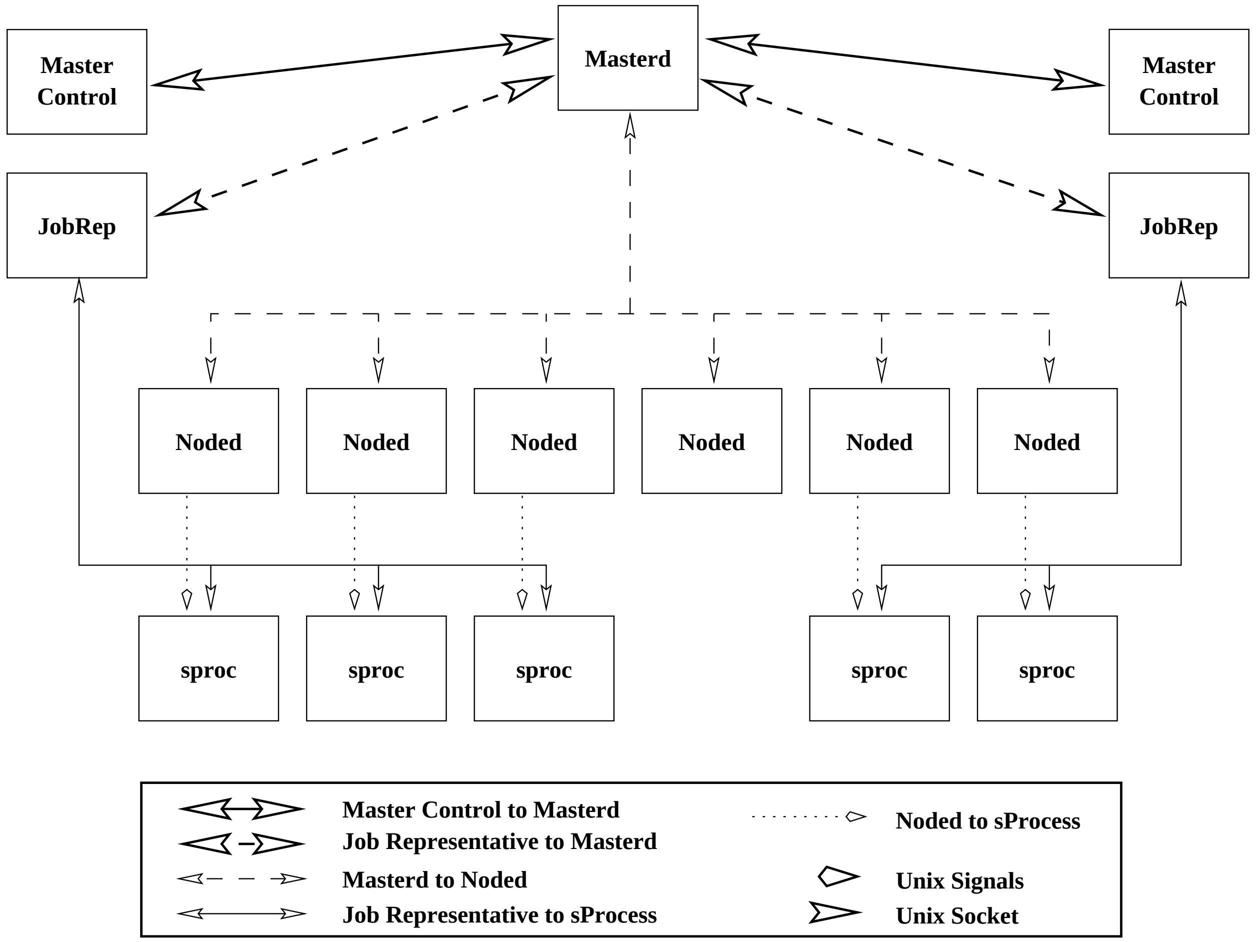


Figure 3: Control Flow

There are, overall, four protocols operating on ParPar:

1. Masterd-Noded protocol. This is the most complex and the most varied in its nature. It includes most of the issues handled by the other protocols, such as job loading, signaling, and system shutdown. In addition, it includes providsions for system booting.
2. Masterd-Job Representative protocol. This protocol deals mainly with job initiation and job termination.
3. Master Control-Masterd protocol. This protocol deals only with system-wide information and control requests — namely, shutting the parallel machine down, requesting the current pool or job configuration, terminating a parallel job, etc.
4. Sprocess-Job Representative protocol This protocol deals only with LED's.

The channels of control flow are shown in Fig. 3. The mechanics of communication are Unix sockets, over the control network, using TCP/IP. In addition, it is possible for the nodeds to send signals to sprocesses running under their control, and sprocesses can cause a SIGCHLD on the nodeds.

10 Parallel Machine Startup

In order to create a parallel machine, some form of "booting" of the same must exist. This booting process links the various independent nodes together and assigns them different roles in the parallel machine. In doing so, it also defines the machine's configuration. The purpose of this section is to define the startup sequence of the parallel machine and to propose a startup protocol.

10.1 Assumptions

The parallel machine is composed of a number of independent machines which, on startup, are aggregated into a two-tiered parallel machine — The "visible" tier is the front-end (master) node, while the "invisible" tier are the two or more back-end nodes. Only the back-end nodes (nodes, hereafter) provide computing resources to the parallel programs. Only the master provides an interface to the nodes.

The booting process has three components:

1. Master Startup
2. Node(s) Startup
3. Initial Configuration Management (Binding)

The first two may proceed in parallel. The third is done after the startups complete.

10.2 Master Startup

The master node, except for running masterd, is a full, autonomous, Unix machine, supporting a full complement of Unix commands, daemons, and environments, etc. Therefore, the master, upon a normal Unix-like startup and as a last step of the same, invokes a daemon (called **masterd**) that is responsible for the parallel computer and represents it to the user. Indeed, a master node is known implicitly by having a *masterd* daemon running on it. A node is, likewise, known implicitly by having a *noded* daemon running on it.

After being loaded, masterd performs its own, internal configuration actions (see Section 17.1). It then creates three sockets and binds them to two well-known ports, in order to allow nodeds, job representatives, and master controllers to establish connections (a separate port is used for each type of connection). The manner in which such connections are handled is described below (see section 10.4).

The well-known ports are made well-known in a number of ways. First, a set of defaults is hardcoded into all the software components. These defaults are:

name	*description*
parpar.cs.huji.ac.il	master
10000	port for job representatives
10001	port for node daemons
10002	port for master controllers

Before using these defaults, the software conponents try to get the ports using the Unix `get_service_by_name` function, with the following names:

name	*description*
parjr	port for job representatives
parnd	port for node daemons
parmc	port for master controllers

These should be defined in `etc/services`, as described in Section 33. Also, the address of the host can be defined by a command line argument or by an entry in the configuration file.

10.3 Node Startup

Unlike the master, nodes are not autonomous and do not provide a normal user with the capability of using them as standalone machines. Indeed, the nodes are normally invisible to the user. Therefore, nodes do not maintain a full set of Unix daemons, etc. The following services and daemons may be safely omitted:

1. bind — domain name service daemon
2. gated/routed — routing daemon
3. lpd/lp — printer daemon

4. sendmail/smail — the mail daemon
5. xdm/X11 — X11 server
6. ypbind — NIS daemon
7. Some inetd services:
   (a) fingerd — finger daemon
   (b) ftpd/tftpd — FTP and trivial FTP daemon
   (c) identd — user IDENT daemon
   (d) ntalkd — talk daemon
   (e) pop/imap — Post Office Protocol daemon
   (f) rexecd — remote exec daemon
   (g) rlogind — remote logins daemon
   (h) rshd — remote commands daemon
   (i) uucpd — UUCP daemon

Instead, nodes run a node daemon — **noded** — which is responsible for the communication with masterd and with the parallel processes on the node itself.

The noded starts by reading its configuration file, and then it binds to the masterd, as described next.

10.4 Initial Configuration Management (Binding)

After masterd has been invoked on the master node, it waits to accept connections on its well-known ports. Initially, the most meaningful connections are from node daemons. Such connections are immediatly followed by the sending of **capability messages** (see section 26.1).

Nodes which have already loaded their noded, send these **capability messages** when the nodeds finalize their initialization. As nodes that do not bind to the master are useless, they continue to send these messages until acknowledged by the master. However, these re-transmissions are separated by a suitable time period so as not to flood the network and overload the master.

Once a master receives a capability message from the node, it acknowledges its receipt, thus informing the noded that it should cease transmitting capability messages. Based on the capability messages it receives, the master builds a picture of the parallel machine as a whole — namely, number of nodes, their identities and network addresses, their types (master, node, I/O node, etc), and capability data (disk space available on each, memory available, processor speeds, etc). This picture is termed the **pool**.

> *To be determined:* An open question is whether we need to propagate some of the pool information to nodeds. Current thinking is no — nodes are supposed to

be as independent as possible, so they will only get whatever information they need when jobs are loaded. However, if the communication subsystem requires significant data about the participating nodes, it may be more efficient to send the data about *all* nodes once, and then just send pointers into it with each job. This has the obvious disadvantage that all the nodes have to be updated when the configuration changes, so we do not do it for now.

Once this update has been completed, the parallel machine has been established and is ostensibly ready for use.

11 Parallel Program Loading

On ParPar, parallel program loading entails a number of steps, initiated by the user who executes the parallel program. The agent that is used for this purpose is the Job Representative (see Section 19).

11.1 Connect to Masterd

The job representative, upon being started, establishes a connection to the master daemon via a well-known port. It stays connected throughout its lifetime.

11.2 User Load Request

When the user specifies a job to be executed, the job representative sends a message to masterd and informs it of the parallel load request and the resources requested (currently, only the number of nodes).

Additional information passed with the load request is the executable that is to be run, its arguments, the user's environment, and a set of ports on the job representative to be used for standard I/O. In addition, the request may also indicate that the job should be run under debugger control; see Section 30.

The part of the user's environment that is passed is under user control, as described in Section 19.1. By default, the following parts will be passed:

PATH — to enable file access without full paths.

PWD — to enable file access relative to the working directory.

DISPLAY — for use by X.

11.3 Resource Allocation and Job Configuration Update

Masterd checks the resources as listed in its pool. If these are sufficient, the requested resources are allocated (see Section 17.3) and a unique XPID is generated. The job configuration is then updated on the master and transmitted to the affected nodes. This is used by the nodes to get the configuration information that the sprocess may need: the total number of nodes, and its rank. It may also be used by the communication subsystem to support the parallel application.

The information contained in the **job configuration update** are the newly allocated node numbers, their network addresses, and their xpid-internal logical mapping (alternatively, network addresses could be sent once when the system configuration changes, rather than being resent for every job). This allows the communication subsystem to map logical message addresses to physical node addresses. When gang scheduling is used, the job configuration also includes the scheduling slot to be used.

Once nodes are allocated, the number of nodes is stored and forwarded to the Job Representative as part of the load akcnowledgement. This is important in cases when the user specifies a range of options regarding the number of nodes to use.

11.4 Load

Masterd informs the allocated nodes that a load request has been issued, possibly with a request for debugging support. The load image may be transfered in either of two ways:

1. Loading on demand
2. Pre-loading

Initially, we will use loading on demand. This means that the nodes access the image via an NSF mount, and we trust Unix to take care of the details. The other option is to first copy the image to the local disk. Obviously, this option can only be used if all nodes have enough disk space (as indicated by their disk capability in the pool). Two implementations are possible:

- Each noded copies the remote NFS-mounted file to a local file independently of the others. This is expected to become inefficient as the number of nodes grows.
- One node copies the file via NFS, and simultaneously broadcasts the data to the others using the ParPar high speed internal network. This will have to wait for the implementation of broadcasts on that network.

In either case, nodes load the image by fork/exec of the noded, with the following steps:

1. Noded performs a `fork` to create the new sprocess; all subsequent steps are performed only by the child (new) process.

2. The child sets up the execution environment for the sprocess, including:
   - Setting the uid to the uid provided in the load instruction.
   - Copying the specified arguments and constructing an `argv` array.
   - Copying the user's environment
   - Connecting to the job representative via sockets, and redirecting stdin, stdout, and stderr to these sockets.
3. If debugging was requested, the first child process `exec`s the local debugger, which then `fork`s again to create the sprocess.
4. The child `exec`s the sprocess load module, thus running it in a relatively normal Unix environment (i.e. under the user's normal Unix user ID, with NFS access to the user's normal files, and with stdio piped to the job representative on the user's workstation).

Once the parallel program is loaded, each of its parts (subprocesses) is executed on one node allocated by the masterd. Initially, communications between the sprocesses is the responsibility of the programmer, with a little bit of help from the system: the list of node network addresses. Later, support for a user-level implementation of the PVM and MPI message-passing interfaces will be added.

11.5 Error Conditions

Loading a job is especially susceptible to various error conditions. If any occur, the job is aborted and an *informative* message is given.

11.5.1 Authorization

Users who are not authorized to use the system cannot run parallel programs. In this case the load request will fail. Authorization is represented by login capability on the host.

11.5.2 Lack of Resources

If the system does not have the amount of resources requested, the load will fail. If the system is only temporarily out of resources, due to allocations to other jobs, the request will be queued and a message to this effect given. The user can then choose to either leave the job for future execution, or remove it from the queue.

11.5.3 Problems in setup

When setting up the environment for the sprocess, the noded has to set the UID, GID, and change to the correct working directory. In principle, any of these operations may fail, although they are not expected to.

11.5.4 Fork/Exec Failure on Nodes

If any of the nodes fails in the fork/exec sequence, the whole job is aborted and a message about the specific failure is given. Causes of failure include

- Executable not found
- Lack of Unix resources, such as file descriptors, memory, or processes

It is expected that the most common failure mode will be problems with the NFS mounting of the user's file system in order to obtain the executable. Such problems can be identified by one node failing to find the executable while others do find it. This problem will be diminished if and when pre-loading is implemented.

The problem with reporting `exec` failure is that it requires communication from the sProcess to the noded. If all is well, `exec` does not return, and there is no way to send a message to the effect that "all is well". But if it fails, the child process remains a clone of the noded. It then opens a special communication channel to the noded, to send a fail message. Noded, upon receiveing such a message, reports to masterd. The failed clone then exits.

The communication channel uses the `parfail` service provided by the noded. An entry in the `/etc/services` file is required to define this, as described in Section 33.

12 Parallel Programming Environment

A central goal of the project is to keep the programming environment as close as possible to established standards, in order to make it easy for users to use the system. This means that in general applications operate in a normal Unix environment, with well-known communication mechanisms (Unix sockets in the initial phase, PVM and MPI thereafter).

12.1 Unsupported Stuff

However, some things that conflict with ParPar services (mainly in the area of job control) are replaced.

12.1.1 Unsupported Unix System Calls

All Unix system calls concerned with process management will not be supported; if used, the outcome and effect on the system are undefined. This includes the following system calls:

- fork
- exec
- wait
- kill

Other calls are less harmful but still unrecommended:

- alarm
- sleep
- signal
- pipe

However, it is possible to set things up to receive a signal sent by the user from the job representative, and also to handle SIGFAIL which indicates that an sprocess has terminated abnormally.

12.1.2 Unsupported PVM Functions

We will (at least initially) not support PVM functions concerned with configuration, i.e. adding or deleting a node or a process to the job. Only a straightforward SPMD style will be supported.

12.2 Added Stuff

Also, some new things are needed to make use of ParPar's features.

12.2.1 Obtaining Configuration Information

In the initial phase, the only communication mechanism available to applications is Unix sockets. Therefore applications need to know of their configuration. This is done via two functions:

`get_config( proc_num, my_id )`
`int *proc_num` // number of processors used by job
`int *my_id` // logical ID of this process, in range of 0 to `proc_num`-1

`get_map( ip_addrs )`
`int ip_addrs[]` // array with `proc_num` entries

12.2.2 LED Output

LED output is a new type of standard output that is ment to augment and partially replace the need for textual output. The following functions are provided:

`get_led_num()`
returns number of LEDs per sprocess

`set_led_num( num )`
`int num` // requested number of LEDs per sprocess

```
#include <led.h>
led_on( led, color )
int led // which LED, zero based
int color // what color, from the include file
```

12.2.3 User-Defined Log Entry

It is possible to add user-defined entries to the job log using the following function:

```
log_write( str )
char *str // entry text
```

A time stamp is added automatically by the system.

12.2.4 Access to Pentium Counters

The Pentium processor has a number of user-accessible counters that are useful for performance debugging (they count things like cycles, memory stalls, etc.) [5]. Functions to access these counters are desirable.

13 Parallel Program Termination

A program or sprocess may terminate for a number of reasons, specifically

1. Normal termination (program completion)
2. Abnormal termination (failure)
3. User-initiated termination

Each of these results in different behavior vis-a-vis noded-masterd interaction and the state of the parallel machine as a whole. Parts of the interactions are mediated by **meta-signals** (explained in Section 16.1).

13.1 Termination Status Identification

A major problem is how to distinguish normal and abnormal termination of an sprocess. There are a few options:

- Base the decision on the exit status. It is a well known Unix convention that a 0 exit status signifies successful completion, whereas any other status means failure of some sort. The problem is that most programmers do not bother to explicitly return 0 from main (or use `exit(0)`). Thus any sprocess that terminates with a non-zero status may cause the system to signal the other sprocesses and kill them.

- A wrapper can be placed around `main`, and always return 0. Thus any non-zero status will indicate that the program really exited abnormally.
- Use the `wait4` system call and analyze the returned status information. If the sprocess terminated by calling `exit`, even with a non-zero velue, all is deemed well. If it was killed by a signal, this is an abnormal termination. An sProcess can cause the whole job to terminate abnormally be calling `abort`, which sends a SIGABRT signal to itself; this will be propogated to the other nodes, as explained below.

Using the `wait4` system call is the preferred approach. It has the additional benefit of also returning the sprocess's resource usage.

13.2 Normal Termination (Program Completion)

When an sprocess terminates, the noded that invoked it receives a SIGCHLD signal. Calling `wait4` then returns the PID of the terminating process, and its exit status. Assuming the exit status is a success, the normal termination procedure outlined below is invoked.

It is to be noted that program termination may be considered to be composed of two parts — sprocess termination and xprocess termination. The former constitutes a partial termination at best, but since it affects the resources of the parallel machine, it has to be handled.

13.2.1 Sprocess Termination

This process is handled in a number of steps:

1. The Noded gets a SIGCHLD signal, and identifies the cause to be normal termination (using `wait4`).
2. Noded informs masterd that the sprocess has terminated. This notification takes the form of a METASIGCHLD.
3. Masterd updates its job configuration data.

    > *To be determined:* The question of whether or not this should be propagated to all the ramaining nodes depends on the communication subsystem. If it can find out for itself that communication partners do not exist any more, job configuration does not have to be updated explicitly. This would have the added advantage of avoiding race conditions when multiple sprocesses terminate at about the same time.

4. Masterd informs Job Representative of the sprocess termination and its exit status. In the future we may aggregate such messages to reduce overhead when many sprocesses terminate at about the same time.

13.2.2 Xprocess Termination

If and only if the sprocess was the last component of an xprocess, masterd decrees that the whole job is done. It then logs information about the job, performs cleanup operations, and informs the Job Representative of the xprocess's termination.

13.3 Abnormal Termination (Failure)

Abnormal termination is related to a single sprocess. There is no notion of global xprocess failure. The primary difference between a normal and an abnormal termination is the behavior of the parallel machine towards the offending xprocess. Specifically, masterd is responsible for reflecting the resulting status information to all sprocesses that are members of the xprocess.

The steps in this process are:

1. The Noded gets a SIGCHLD signal, and identifies the cause to be abnormal termination because of a signal (using `wait4`).

2. Noded sends a METASIGFAIL to masterd, with the cause of termination.

3. Masterd reflects the METASIGFAIL to other member nodes.

4. Each noded process the MATASIGFAIL and sends a SIGFAIL signal to its member sprocess (SIGFAIL will be one of the common Unix signals that cause a process to die unless it is caught, possibly SIGINT). If any sprocess terminates because of this SIGFAIL, the noded informs masterd about its termination. This is done by a METASIGDIED, which is distinct from both METASIGCHLD and METASIGFAIL: it is not a normal termination, but it need not be reflected.

5. Masterd collates the signals and updates its job configuration data. For each terminated sprocess, masterd notifies Job Representative that an sprocess has failed or been killed bacause another had failed. If all sprocesses have terminated, masterd notifies the Job Representative that the xprocess has terminated due to an sprocess failing (and provides the reason thereof, if any).

A special case occurs when an sprocess decides to terminate the whole xprocess using the `abort` system call. This causes a METASIGABRT to be sent to all sprocesses, rather than a METASIGFAIL.

13.4 User-initiated Termination

In effect, user-initiated termination is the simplest to process — the various daemons simply transmit the request down to the xprocess components which, thereupon, are supposed to suicide.

The steps are:

1. Job Representative informs masterd that its xprocess is to be terminated.
2. Masterd transmits METASIGKILL to the affected nodes.
3. Each noded issues a SIGKILL to the affected sprocess and informs masterd of successful completion.
4. Masterd updates its job configuration data (essentially returning the nodes to the free pool) and informs Job Representative of the termination of the xprocess.

The user may also send other signals, not only SIGKILL; they are propagated using a METASIGSIG, as explained in Section 16.

14 Parallel Program I/O

Three types of I/O requests are considered:

- File I/O
- Standard I/O
- LED output

14.1 File I/O

Currently, file I/O is considered to be the responsibility of the user. This is supported by NFS mounting the user's file system on the master and all of the nodes. In the future, a parallel file system will (hopefully) be implemented.

14.2 Standard I/O

Standard I/O is redirected to the Job Representative for display to the user. This bypasses the masterd and noded's for efficiency, as shown in Fig. 4.

The noded sets things up as follows. After `fork`ing the sprocess, but before `exec`ing it, the new child process (the one that will soon `exec` to become the sprocess) opens 3 sockets to the job representative: one for stdin and stdout, one for stderr, and one for LEDs. It then redirects all standard I/O from/to these pipes.

On the job representative side, anything coming up the stream to the job representative is tagged with the job node ID number and displayed; stderr is displayed in red. Typically the job representative will not break a stream and interleave it with output from other nodes, but this may happen due to Unix buffering. Using `fflush` should guarantee that individual writes are not broken. For stdin, the Job Representative sends data to the relevant sprocesses, as identified by a prefix to the input.

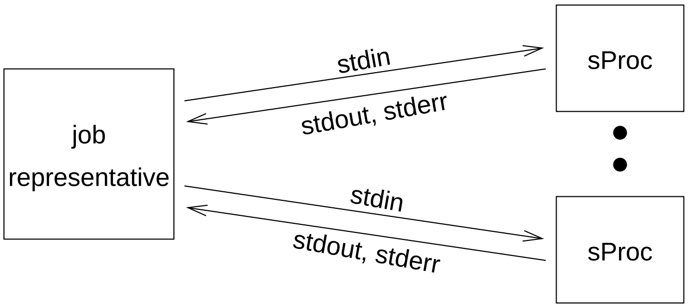


Figure 4: Communication channels used to implement standard I/O.

This scheme ensures that from the sprocess point of view standard I/O behaves normally, as in any Unix system, using the same file descriptors. Thus normal I/O calls from C and Fortran should work.

14.3 LED's

LED's are inspired by (and named after) the little colored lights that serve as status indicators on the front panels of computers. Instead of real LED's, these are little squares on the screen that can be colored under application control, providing an additional output mode that may serve as an alternative to conventional textual I/O [10]. Each node is represented by a fixed number of LEDs (by default, one), which can also be set by the program.

The LED functions (see section 12) are implemented in a library that is linked with applications running on ParPar. These functions use the socket set up by the noded before the sprocess was `exec`ed to communicate with the job representative. The file descriptor for the socket is passed to the library routines via the `LEDSOCK` environment variable.

15 Parallel Machine Fault Handling

Regrettably, due to the fact that a parallel machine is composed of a number of relatively independent components, we need to handle cases in which some part of the machine fails to deliver. First, there is a need to identify such situations. Then, there is a need to decide what to do about it.

Note that this section deals with *system failures*, i.e. cases where a node goes down or one of the system daemons fails. Failure of an sprocess is discussed in Section 13.3.

15.1 Heartbeat

The parallel computer will maintain a heartbeat between its nodes and the master. In order to limit the required network traffic for what amounts, after all, to an ancillary function to minimum, the heartbeat shall be both implicit and explicit.

Explicit heartbeat will take the form of a short packet sent from the master to the nodes in a round-robin schedule every minute or so. The node, on receipt of a heartbeat packet, must reply.

To reduce the overhead of maintaining a heartbeat, an implicit heartbeat is also maintained — any network packet received from a node is, in effect, an implicit heartbeat packet. Therefore, any node successfully transmitting any packet to the master shall be stricken from the round-robin scheme for a single pass.

It should be noted that the TCP protocol suite maintains a heartbeat as well. Therefore, if we use TCP for control communications (which we intend to do), an explicit heartbeat may be unnecessary, because TCP will flag any connection that goes down. However, we may still want our own heartbeat, as we want to be notified fo failure much sooner than what TCP provides.

To be determined: Is it possible to tune TCP in this respect?

15.2 Node Failure

The heartbeat mechanism will identify node failures sooner or later, and so they will come to the attention of the masterd.

Masterd, upon learning that a node has failed, performs a system configuration update. Affected xprocesses are identified, and other members (i.e. sprocesses) in these xprocesses are notified via a SIGFAIL signal, as when an sprocess fails (see section 13.3).

Thus the effect of a node failure is limited to those jobs that use the failing node. It should have no effect on the rest of the system.

Partial node failure (e.g. noded failing, but the node itself still functional) is deemed to be equivalent to the node being down. If possible, some mechanism for establishing the nature of the partiality of the failure must be identified.

15.3 Master Down

When the host node is down, or the masterd fails, the whole system is deemed down — it is impossible to submit any more jobs or perform control functions.

Note, however, that jobs that are already running are actually independent of the master except for termination control. Therefore such jobs may continue to run to whatever degree possible. This requires two modifications to other parts of the design:

- When an sprocess terminates, the noded normally notifies the masterd. If the masterd is not accessible, it should notify the job representative, and the job representative should be able to collect such information and note when the whole job has terminated.

- When the masterd restarts, it should issue a bind request, the nodeds that are still up and running previous jobs should be able to notify the masterd of this status, so that these jobs will not be affected.

This whole issue is currently left for the farther future.

15.4 Online Configuration Management

When masterd identifies a node as down, it performs a system configuration update and job configuration update, as described above. The same happens when a node is taken down by the superuser.

When a node is fixed and it comes up again, its noded sends a capability message to the masterd. Masterd then incorporates it into the pool. Thus there is no need for an explicit interface to add nodes to the system.

16 Signals and Exceptions

Using the signal suite provided by Unix seems to be a logical and simple choice for controlling a parallel machine. The mechanism is well understood and much described, provides an abundance of signals and has a certain (albeit a sharply limited one) scope for expansion.

However, in a parallel machine, it is not at all certain that a full suite of signals provided by Unix is needed or useful. It is, therefore, of importance to decide which signals are usable by the parallel computer. In addition, mechanisms for sending a signal to a group of sprocesses on different nodes are needed.

16.1 Signals and Meta-Signals

A signal may originate for a large number of reasons. In the context of a parallel machine, however, it seems advisable to sharply and clearly delineate not the *reason* for a signal (for a reason for a signal on one node does not necessarily pertain on another node) but its *scope.*

Thus, we divide the signals into two, conceptually different but functionally equivalent, strata — normal Unix signals that pertain to a single sprocess, and *meta-signals* that pertain to a whole xprocess. As job representatives and nodeds are not connected to each other directly, meta-signals pass through masterd.

Meta-signals originating from a node are (to save both network bandwidth and, especially, complexity) transmitted only to the masterd. If so required, the masterd is responsible for "bouncing" or "reflecting" the said signal to the rest of the nodes. This may be readily accomplished by a multicast or a broadcast operation, an added advantage since the nodes in such a configuration, need not know about each other. In other cases, the masted is the sole recipient of the meta-signal.

A meta-signal originating from the master is simply broadcast or multicast to the nodes. In this respect, the master may act as agent for the job representative, who is the true source of the signal. Upon arriving at a node, the meta-signal is typically translated into a normal Unix signal that is applied to the relevant sprocess.

16.2 Signal Semantics

Unix signals are used for many different things. Here we try to cover most of them.

16.2.1 Exceptions

A large set of signals are used by the operating system (Unix) to tell the process about an exception in its execution. The process may catch these signals and handle the exception, but if it does not, the default action is usually to terminate the process. This set of signals includes

- HUP - hangup
- ILL - illegal instruction
- TRAP - trace trap
- EMT - emulator trap
- FPE - arithmetic exception
- BUS - bus error
- SEGV - segmentation violation
- SYS - bad argument to system call
- PIPE - broken pipe
- XCPU - CPU time limit
- XFSZ - file size limit
- LOST - resource lost

We extend this concept and include “node failure” as a new signal for xprocesses. Thus if the sprocess on any node fails (possibly due to one of the above signals), or the node itself goes down, the *other* sprocesses will get a SIGFAIL. This conveys the information that an sprocess has gone down. If the user writes an appropriate handler, it is then possible to recover from such failures. If not, then the other sprocesses will be killed (the default action for SIGFAIL), and the whole xprocess will terminate.

16.2.2 Control

Other signals are used to control processes. These may cause the process to terminate or to stop if they are not caught (and not all may be caught). The list includes

- INT - interrupt
- QUIT - quit
- ABRT - abort

- KILL - uncatchable kill
- ALRM - alarm clock
- TERM - software termination
- STOP - uncatchable stop
- TSTP - stop from keyboard
- CONT - undo a stop
- TTIN - background read attempt
- TTOU - background write attempt
- VTALRM - virtual time alarm
- PROF - profiler alarm

These signals are problematic and should not be used directly on sprocesses, job representatives, etc. However, they are used internally by the system (e.g. SIGFAIL will most probably actually be a SIGINT), and some of them, most notably KILL, are available as meta-signals. Using KILL directly may leave the system in an inconsistent state. It is a challenge for the future to design a fix for this problem.

An important control signal that will possibly be added in ParPar is the one that will cause an xprocess switch — the gang scheduler signal. This may be implemented by a meta-signal that does not have a direct match in a normal Unix signal, but rather translates to a pair of such signals, used to stop the running process and start another.

Note that signals used to debug normal Unix processes will continue to be used in their normal function. In this way the distributed design of our parallel debugger leverages on existing technology, and we do not need any new control mechanisms at this level. The only change is in controlling the debugger processes, and telling them what to do.

16.2.3 Information

The last set of signals conveys information. We do not have plans for these at the moment.

- URG - urgent situation on socket
- CHLD - child status change
- IO - I/O possible
- WINCH - window changed

16.3 Meta-Signals

Meta-signals are used in a number of cases:

1. When an sprocess terminates (normally or abnormally), its noded sends a meta-signal to masterd.

2. In case of abnormal termination, the masterd reflects a meta-SIGFAIL to the other sprocesses in the job.

3. The user may send a meta-signal from the job representative.

The signals available currently are:

METASIGCHLD A signal sent from a noded to masterd when an sprocess terminates normally.

METASIGFAIL A signal sent from a noded to masterd when an sprocess fails for any reason. It is then reflected by the masterd to nodeds with other sprocesses in the same xprocess. Such a signal is also initiated by the masterd if a node fails completely (in which case the relevant noded can't initiate it).

METASIGABRT A signal used in order to propagate a SIGABRT signal, and allow one sprocess to abort the whole xprocess. It is sent from the noded of the aborting sprocess to masterd, and then reflected to other nodeds as above.

METASIGKILL A signal sent from a job representative to masterd, and then propagated to all nodeds with sprocesses belongin to this job. It is used to kill these sprocesses, and, by inference, the whole xprocess.

METASIGDIED A signal used by a noded to report to masterd that an sprocess has dies as a result of a SIGFAIL, SIGKILL, or SIGABRT. This should *not* be propagated to other nodes, as it is assumed to be the result of a previous signal that was propagated.

MATASIGSIG Used to send any regular Unix signal (other than SIGKILL), from the job representative, through masterd and the nodeds, to the job's sprocesses.

METASIGSWITCH A signal generated by masterd to initiate an xprocess context switch at the nodeds.

Part IV
The Software Components

The procedures described above are carried out via interactions among a number of software components. In this part, these components are described. They include the master daemon, the node daemon, user interfaces for job control and system control, and the application library.

17 The Master Daemon (masterd)

The master daemon is a central component of the system, and serves multiple functions. Its main independent function is configuration management, and using the configuration information for resource management. In addition, it serves as a bridge between the job representatives and their nodes, and between the nodes themselves when necessary (e.g. in case of a node failure).

17.1 Initialization

Upon being loaded, masterd performs its own initialization. This includes reading the system configuration file, initializing data structures, and establishing sockets for services.

17.1.1 Configuration File

This file provides the following information:

- Listing of nodes with special functions, such as I/O nodes. Any unlisted nodes that attempt to bind are considered to be plain compute nodes.
- Listing of users with special privileges, i.e. "superusers" for the purposes of the parallel system only.

17.1.2 Data Structures

The master daemon maintains three main system tables:

- `nodetab` contains information about compute nodes. It is indexed by the socket file descriptor used to communicate with the node. During initialization, all entries are marked as unused.
- `guitab` contains information about GUIs, namely job representatives and master controllers. It is indexed by the socket file descriptor used to communicate with the GUI. During initialization, all entries are marked as unused.

- `xpidtab` contains information about jobs (that is, xprocesses, which are identified by a system-unique xpid). An entry is allocated as each job is loaded. During initialization, all entries are linked in a free list and marked as unused.

17.1.3 Sockets

Finally, the master daemon creates a set of three sockets (with the `socket` system call), `bind`s them to the three services it provides, and `listen`s on them. These are the well-known ports that are used by other components. The three services are for connections from nodeds, job representatives, and master controllers.

17.2 Configuration Management

One of the main responsibilities of the master daemon is keeping track of what goes on in the system. This includes two types of configuration management: system configuration management, and job configuration management.

17.2.1 System Configuration Management

System configuration management means knowing which components are up and participating in the system. This comes in two flavors: compute nodes which bind to the masterd and serve sprocesses, and GUIs that represent users.

The binding process used to establish the initial contact between the master daemon and node daemons is described in Section 10.4. Note that this process is initiated by the nodes, so nodes may enter the system at any time by sending a capability message. Upon receipt of such a message, the node is entered into the `nodetab`. The entry is deleted if and when the node fails (as detected by the heartbeat mechanism — Section 15), or when it is deleted explicitly by the superuser.

No.	IP addr	type	status	CPUs	arch	clock	memory	disk

subpool information

Figure 5: The pool and subpool data structures.

The nodetab, also known as the **pool**, contains all the available information about the current configuration (Fig. 5). This is raw data about the identity of the node and its capabilities. Information about the actual load on the nodes is maintained by the resource manager (which is a separate component within the masterd — see Section 17.3).

If the communication subsystem requires this, the master may create a **sub-pool** data structure, composed of only the node-ID's, the node types and their network addresses. This sub-pool will be transmitted to the nodes, thereby providing the nodes with a unified picture of the parallel machine. Initially, a small number of such messages, each with information about many nodes, can be expected. As nodes can update their configuration incrementally, the system does not require that all nodes come up and bind at exactly the same time. Later, the same type of messages are used when single nodes are added or deleted. However, the current design does not require such subpool information.

GUIs are handled similarly to nodeds. When a GUI first establishes a connection with the masterd, this is recorded in the `guitab`. This entry is maintained until the GUI terminates; it is needed in order to send system messages to users.

For all GUIs, the uid of the user they represent is maintained. This is used to verify user permisions when various requests are submitted (e.g. to kill a job). Additional information (e.g. the entry in the xpidtable of a job represented by a job representative) is recorded as needed.

17.2.2 Job Configuration Management

Job configuration management means keeping track of the status of jobs and all their constituent sprocesses. Jobs initially get a set of nodes, and this is not expected to change. However, some sprocesses may terminate before others, and nodes may also fail in general. Therefore it is prudent to maintain information about the jobs in the system.

The main information maintained for running jobs is the xpid and a bit-vector identifying the nodes with sprocesses belonging to this job. This is needed in order to handle meta-signals that need to reach all sprocesses, e.g. METASIGFAIL and METASIGKILL (see Section 16). Counters are used to keep track of the loading of sprocesses and their termination. Also, accounting information is maintained and then written to the accounting log when the job terminates (see Section 29.1).

17.3 Resource Management

Resource management is the function of allocating nodes to jobs so they can run. In the initial version, an adaptive partitioning scheme will be used. Later, this will be extended to support gang scheduling.

17.3.1 Adaptive Partitioning

In adaptive partitioning, jobs get a partition of the machine for their exclusive use. The size of the partition is determined by the system based on the job requirements and the current system load [20]. In our case, the user can express the requirements by giving a set of possible sizes or a range of sizes. The interface is described in Section 19.2.

It is to the user's advantage to specify as wide a range as possible. Thus if only a few nodes are available, the job will nevertheless be able to run. If on the other hand many nodes are available, the system will know that they can all be used and allocate them. Had only a single size been specified, the job would not be able to run if less are available nor make use of more nodes if they are available.

The algorithm for adaptive partitioning is based on the notion of a **target partition size**. The target partition size is defined to be the number of nodes in the system divided by the number of jobs in the system. Whenever nodes are available and jobs are queued, the system attempts to make an allocation. This is done be creating a partition size that

1. is as close as possible to the target partition size
2. is one of the sizes requested by the job
3. is not larger than the number of available nodes

This is done iteratively for the jobs in the queue, until no more allocations can be made.

Note that this algorithm allows jobs that require few nodes to leapfrog jobs that need many nodes, leading to a danger of starvation. The solution is to define a maximal delay for each job, proportional to the quotient of its minimal requirement and the target partition size (so jobs that require more may be delayed more). Each time the job is leapfrogged, the delay counter is decremented. When it reaches zero, no more jobs are allowed to leapfrog it, and it waits at the head of the queue until a sufficient number of nodes becomes available. Note that we do not have information about the expected runtime of jobs, so algorithms such as backfilling cannot be used [16].

17.3.2 Gang Scheduling

Gang scheduling introduces significant flexibility to allocation, so having a wide range of options becomes less important. However, having some flexibility still allows the system to pack jobs together more efficiently.

With gang scheduling, resources are viewed as a matrix, where columns are nodes and rows are **scheduling slots** [19]. The sprocesses of each xprocess are allocated to the same slot across the required number of nodes. All nodes perform the context switching from one slot to the next in unison, thereby implementing gang scheduling. For implementation details, see Section 31.

17.3.3 Independence of the Resource Manager

The resource manager is implemented as an independent entity within the master daemon. This means that it maintains its own data structures, and is invoked by a well-defined set of functions. This will allow different resource managers to be written and used interchangeably.

17.4 Job Control

Three job control functions are provided: notification of node daemons about loading new jobs, notification of node daemons about signals, and propagation of signals among the nodes of a job. These are described in Section 11 and Section 16.

17.5 Masterd Control

A user interface to the master, allowing users to examine the pool, the logs, the current job configuration information, and the superuser to control masterd's behaviour and to shut it down, shall be available, in the form of a GUI. See Section 20.

17.6 Code Structure

After initialization, the masterd goes into a control loop where active sockets are `select`ed. Connections and messages may come from nodeds, job representatives, and master controllers. Depending on the source of the message, the masterd enters a switch to handle the specific request.

The main files used are as follows:

masterd.h — definitions of global variables, e.g. system tables. The types used are defined in common.h in the general include directory.

prototypes.h — function prototypes.

masterd.c — main: parsing command line arguments, and general flow.

selfconf.c — handlig the configuration file and initialization.

conmgr.c — the main control loop, and opening sockets for new connections.

node_config.c — managing the nodetab.

nd_switcher.c — switch on messages from nodeds.

gui_config.c — managing the guitab.

gui_switcher.c — switch on messages from GUIs.

rm.c — the resource manager and its private data structures. A number of versions of this file are expected.

jobs.c — non-resource-management job-related functions. This includes a service routine to send a message to all sprocesses, handling of sprocess termination, etc.

Generally speaking, the masterd is event-driven. It waits for messages, and acts upon them. A received message may cause internal state to be updated, and one or more messages to be transmitted. However, the design in non-blocking, in the sense that the masterd will never block waiting for an additional message to arrive. Instead, each message is handled independently.

18 The Node Daemons (noded)

The node daemon (noded) is the masterd's counterpart — it is responsible for the binding process, for creating sprocesses when a program is loaded, for meta-signal conversion, and other sundry tasks.

18.1 Noded Startup

The node daemon is started automatically at the end of the Unix boot process on each node. It then reads its configuration file, which may include information about the address of the host where the masterd is running. Finally, it initializes its system tables.

18.2 Binding Mechanism

Nodeds bind to the parallel machine by sending a capability message to the master daemon (see Section 10.4). This message is re-transmitted until an acknowledgment is received from the master daemon. There are no timing constraints: the masterd may come up before the nodeds or after them, the nodeds may all come up at the same time or they may come up at different times, etc.

The noded may have to keep a data structure with information about the whole system configuration, called the subpool, but this is not envisioned in the current design.

18.3 Job Loading and Termination

When a parallel job is loaded, an sprocess must be loaded on each node. This is described in detail in Section 11. Using a resident daemon to load sprocesses and control them has been done before, e.g. the `startd` daemon in Condor [17], and is more efficient than using mechanisms such as `rsh`.

When an sprocess is forked, its pid is stored in the `sproctab`. The slot used is the same on all nodeds, and is actually assigned by the masterd as part of loading the job. Thus the noded does not have to manage this table. This is done to ease teh support for gang scheduling, when it is implemented.

When an sprocess terminates, the noded gets a SIGCHLD. It then uses the `wait4` system call to learn what happened, and sends the appropriate metasignal to the masterd. The options are:

METASIGCHLD - normal termination

METASIGFAIL - abnormal termination as a result of a signal

METASIGABRT - termination as a result of a SIGABRT signal

METASIGDIED - termination as a result of a signal sent by the noded, such as a reflected METASIGFAIL, METASIGABRT, or METASIGKILL.

To acertain the cause of termination, the noded must therefore record the last signal (if any) that it had sent to each sprocess.

18.4 Signal Translation

Meta-signals destined for a whole xprocess are sent to all nodes with component sprocesses. It is the job of the nodeds on these nodes to translate the meta-signals into normal Unix signals addressed to the relevant sproceses.

18.5 Noded Code Structure

After initialization the noded goes into a control loop waiting for messages from the masterd. However, there may be asynchronous interruptions due to SIGCHLDs when sprocesses terminate. The SIGCHLD handler increments a counter, which is checked in the control loop, and if positive calls a cleanup function. Thus at any time the noded is either serving a message from the masterd or cleaning up after an sprocess.

The control loop may also receive new connections from sprocesses which forked successfully but then failed to exec. These connections are used to relay information about the failure, which is then forwarded by the noded to the masterd.

The main files used are as follows:

noded.h — definitions of global variables, e.g. system tables.

prototypes.h — function prototypes.

noded.c — main: parsing command line arguments, general flow, and the control loop.

selfconf.c — handlig the configuration file and initialization. Also includes the SIGCHLD handler.

bind.c — handling the initial bind process.

md_switcher.c — the switch on the different messages that may arrive from the masterd.

image_load.c — functions to collect the messages that constitute a job load request, and finally to fork and exec it.

cleanup.c — calling wait4 after an sprocess terminates and sending information about the termination to the masterd.

signals.c — forwarding of metasignals to the sprocesses.

19 The Job Representative

The job representative is the interface between the user and his job. It provides facilities for submitting jobs, for standard I/O, and for signaling. It does not provide job control: each representative handles one job at a time. If it is desired to run multiple jobs, then multiple representatives have to be started. Also, the job representative does not provide a debugging interface — this is done be a separate parallel debugger front end.

19.1 Window Layout

The job representative employs a graphical user interface with the following components:

- A **menubar** with the following menus:
  - file menu with the following operations:
    - load and save configuration file. This allows all the information contained in the job representative to be saved for re-load in a future session. Such information includes the application executable, its arguments, resource requirements, and standard I/O characteristics.
    - redirect standard I/O to files. This option is only enabled when a job is not currently running.
    - quit.
  - environment menu which controls what part of the environment is passed to the job. The default is just a few select fields (see Section 11, to reduce overhead, but it is possible to pass the whole environment or select specific fields using a checkbutton submenu.
  - signal menu which allows various signals to be sent to the application. This menu is only enabled when a job is running.
  - window menu which allows various configurations of the standard I/O window, as described below.
  - help menu.
- A **job submittal** panel with entry widgets for the application path name, its arguments, and possible configurations. This is explained below (Sect. 19.2).
- A **system messages** window where various status and error messages are displayed.

- A **standard I/O** panel containing a text window and a LED array. These are explained below in Section 19.3.
- A set of buttons with the following functionalities:
  - load an application as currently specified in the job submittal panel.
  - kill the currently running application.
  - use the debugger when loading the job (this is a checkbutton).
  - be verbose about the job, i.e. report every little thing that happens to it (this is a checkbutton).
  - run the job in batch mode, i.e. low priority (this is a checkbutton).
  - clear all entry widgets.
  - obtain help.
  - close the job representative. If a job is running, this implies disconnecting from the job, which can no longer perform any standard I/O operations.
  - quit the job representative. If a job is running, this implies killing the job; a dialog concerning this action should be displayed to allow the operation to be cancelled.

A tentative layout is depicted in Fig. 6.

19.2 Job Submittal

The process to submit a job for execution is:

1. Enter the path name for the job's executable in the executable entry.
2. Optionally enter arguments that should be passed to the executable in the arguments entry.
3. Enter the number of nodes to be used in the nodes entry.
4. Press the load button.

To run the job under debugger control or in batch mode, the debug or batch buttons must be pressed before the load button is pressed. Likewise, verbose must be pressed to get detailed information about the job (rather than the default summary information).

The ParPar resource manager will use adaptive partitioning (later combined with gang scheduling) to allocate nodes to jobs. This means that the number of nodes allocated will be based on both the user's request and the current system load. Thus the user can enter a list of sizes in the `nodes` entry, rather than only one size. Also, it is possible to press the `nodes` label and obtain a sub-window with help for choosing sizes, including the following options:

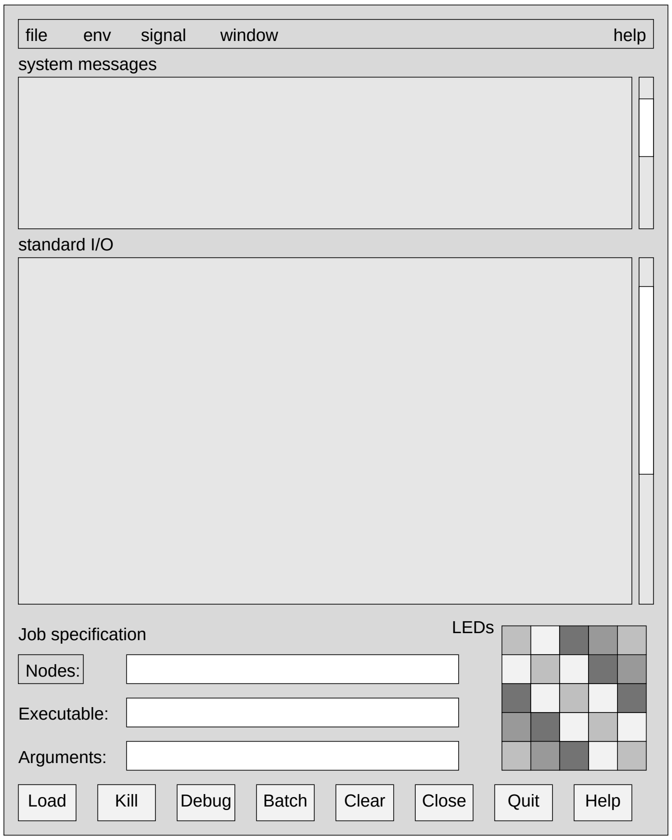


Figure 6: Sketch of window layout for job representative.

- Express a range from a minimal size to a maximal one.
- Selection within the range, including
  - sizes with a given increment.
  - powers of 2.
- a specified list of sizes.

For example, to get the maximal partition size currently available specify a range of 1 to the full machine size, with increments of 1. The system will then select the largest size possible out of this specification, which is the number of currently free processors. Likewise, to select the maximal power of two available, specify the range from 1 to the full machine size with a selection of powers of two.

The details of the resource management algorithms that use this information are explained in Section 17.3.

19.3 Standard I/O

Conventionally, standard I/O includes text input and output to the user's terminal. In ParPar, we will add an additional type of standard output, namely LEDs.

Text input and output is done via a text widget in the job representative window. By default, text from all nodes is shown, interleaved by the order in which it was received, with each line prefixed by its source node number. Text typed by the user is sent to all nodes unless prefixed by a node number. Stderr is shown in red.

Alternatively, the text window can be subdivided into a number of smaller windows, with each assigned to a different node. This allows standard I/O from a subset of nodes to be seen independently, rather than in interleaved form. The number of subwindows is controlled via the window menu.

The LED output is an array of small squares that can be colored in different colors, under program control. Each node is represented by a fixed number of LEDs (by default, one). By changing the color of the LEDs, nodes can signal their status without using text I/O [10].

If many LEDs are used, it may be desirable to enlarge the LED array and position it more centrally (maybe even in place of the standard I/O window). This option will also be available via the window menu.

19.4 Signaling the Job

Sending a signal to a parallel job is done by sending a meta-signal to the master daemon, which forwards it to all node daemons, who translate it into a conventional Unix signal. The most important signal, kill, is provided as a button. Other signals can be invoked via the signal menu. For details, see Section 16.1.

20 The Master Control Interface

The master control interface actually has two versions: for the superuser, and for mere mortals.

20.1 Normal User Interface

The normal interface allows users to obtain information about the system. This is separated from the job representative, because users may have multiple job representatives for running multiple jobs, but only need one system information interface.

the information provided includes:

- Pool information, i.e. the nodes in the system and their capabilities.
- A load histogram showing the load on the different nodes, in terms of number of jobs using each. In the first phase, where adaptive partitioning is used, this can only be 0 or 1. With gang scheduling, higher numbers are possible.

- A listing of submitted jobs (similar to `ps` on Unix). Each entry will include the user, number of nodes, and runtime. In the first phase, it will also indicate if the job is running or queued. Upon request, the list of nodes used by the job will be provided as well.

In addition, it is possible to request that a self-owned job be killed. This is translated by the master controller into a METASIGKILL.

20.2 Additional Superuser Interface

Superusers will have additional power on the system, which may be exercised via the master controller. This includes

- Delete any job from the listing of submitted jobs. This is translated by the master controller into a METASIGKILL.
- Delete a node from the pool. Sprocesses running on the node are given a "grace period" of a few minutes to complete; if they do not, they are then killed. Specifying a grace period of 0 kills all such sprocesses immediately.
- Shut down the system gracefully. Again, jobs running on the machine are given a "grace period" of a few minutes to complete; if they do not, they are then killed. Specifying a grace period of 0 kills all jobs immediately. In either case, additional job submittals are disabled, and once all current jobs are done, all daemons exit.
- Write a message to all users (similar to the Unix `wall` command). This will appear in the "system messages" area of all job representatives.
- Specify the level of logging in the system. See Section 29.2.

21 The Application Library

Applications running on ParPar will have to be linked with an appropriate library. A number of versions will exist over time:

1. Library of LED functions.
2. Library of configuration functions for initial version.
3. Library of MPI functions.
4. Library of PVM functions.

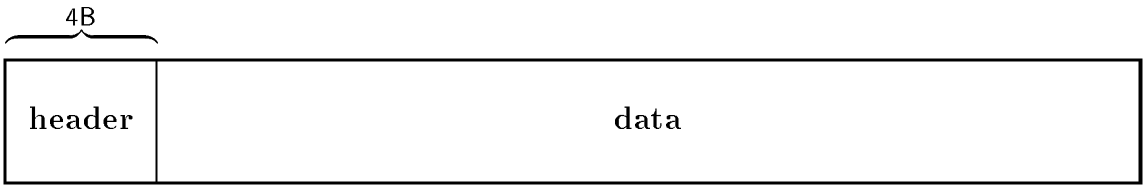
Part V
The Protocols

There are, overall, four control and communication protocols used in the parallel machine:

1. The Masterd-Noded control protocol
2. The Masterd-Job Representative protocol
3. The Master Control-Masterd protocol
4. The Job Representative-Sprocess protocol

A packet may be four or more bytes in length. The length of the packet is encoded in the header and packets are read from the net in two consecutive read calls — the header and then the packet itself.

A packet has the following structure:



The header is an unsigned 32-bit value, structured thus: 0xVABBCCCC, where

V is the protocol version

A is the protocol group:

01 Master Control to Master Daemon control protocol

02 Node Daemon to Master Daemon control protocol

04 Job Representative to Master Daemon control protocol

08 Job Representative to Sprocess control protocol

B is the message number

C is the message size

In the following, the protocol group and message number are indicated for each message described, and the size is not.

All four protocols share a common message numbering scheme. Some packets are forwarded from one protocol to another; for example, a load request. In such cases, the packet structure and message number are the same — only the protocol group changes.

The following description corresponds to the organization of `.h` files that contain definitions of message structures. This is depicted in Fig. 7.

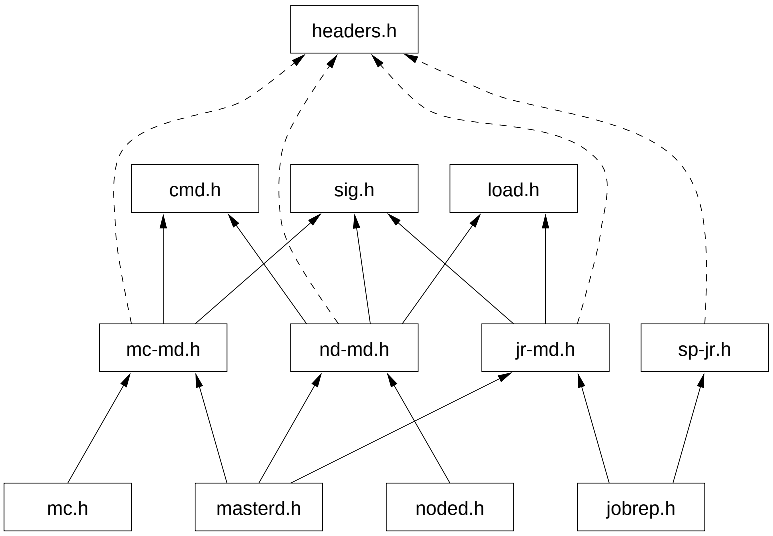


Figure 7: Organization of .h files with message structures.

headers.h — definitions of protocol group numbers and message numbers. These are combined into specific message numbers or each protocol in the four protocol files: mc-md.h, nd-md.h, jr-md.h, and sp-jr.h.

cmd.h — structures for commands issues by the master control interface and forwarded by masterd to nodeds (e.g. shutdown).

sig.h — definition of metasignals. These can originate from any of the components, and are sometimes forwarded too.

load.h — structures related to loading a job. These are originated by the job representative and forwarded to nodeds.

mc-md.h — message number definitions, and structures for non-forwarded commands from the master control to masterd (e.g. show configuration).

nd-md.h — message number definitions, and structures for interactions between masterd and nodeds (e.g. binding).

jr-md.h — message number definitions, and structures for non-forwarded commands from the job representative to masterd (e.g. allocate resources).

sp-jr.h — message number definitions and structures for LED operations.

22 The System Commands Protocol in Detail

These messages are defined in `cmd.h`. They are used in the mc-md and nd-md protocols.

22.1 Define Logging Level

Usage: set the logging level in the system.

See Section 29.2.

Routing: from master controller to masterd. Masterd then forwards this message to the nodeds.

Can only be sent by a superuser.

Structure:

	2B
0x0134 0x0234	detail

detail is the desired level of detail of the logs, as described in Section 29.2. It is expressed as a system constant from syslog.h.

Comments: This overrides the previously specified logging level, and imposes a uniform level on the whole system.

22.2 Parallel Machine Shutdown

Usage: causes the parallel machine to shut down.

See Section 20.2.

Routing: from master controller to masterd. Masterd subsequently forwards this message to the nodeds.

Can only be sent by a superuser.

Structure:

	4B
0x0162 0x0262	grace period

Grace Period is the time allowed for jobs to terminate, before they are killed, in seconds. If 0, they are killed at once.

Comments: the masterd handles the grace period, and then forwards the message to all nodeds. Nodeds kill all jobs immediately, and ignore the grace period.

23 The Meta-Signals Protocol in Detail

These messages are defined in `sig.h`. They are used in all 3 main protocols: mc-md, nd-md, and jr-md.

23.1 Meta-Signal

Usage: send a meta-signal for information or to cause a Unix signal to be sent to a certain job's sprocesses.

See Section 16.

Routing: there are a few possibilities:

1. When an sprocess terminates (normally or abnormally), its noded sends a meta-signal to masterd. See Section 13.2
2. In case of abnormal termination, the masterd reflects a meta-signal to the other sprocesses in the job. See Section 13.3
3. The user may send a meta-signal from the job representative to a given xprocess via masterd which forwards it to the nodeds responsible for the xprocess. See Sections 19.1 and 13.4.
4. The user may also send a meta-signal to a given xprocess from the master control interface. See Section 20.1. Likewise, the superuser may send a meta-signal to any job. See Section 20.2.

In the first case, the noded also uses this message to piggy-back information about resource usage by the terminated sprocess, which is then used by masterd for accounting (see Section 29.1).

Structure:

	4B	4B	4B	1B	8B	8B	
0x0121 0x0221 0x0421	meta signal	status	xpid	slot or xpid ent	user time	system time	...

	4B	4B
...	memory use	disk use

meta-signal is the meta-signal type itself.

status is the exit status, as returned by `wait4`, when an sprocess terminates normally. For abnormal termination, it is the signal that caused the sprocess to terminate. It is also the signal number when a meta-signal is initiated by the user.

xpid is the job to which the meta-signal pertains.

slot_or_xpidentry contains the slot in the sproctab when routed to a noded, and the xpidentry in the xpidtab when routed to the masterd.

user time is the user CPU usage of a terminated sprocess. This field is filled only by a noded upon a termination of an sprocess.

system time is the system CPU usage of a terminated sprocess. This field is filled only by a noded upon a termination of an sprocess.

memory use is the memory usage of a terminated sprocess. This field is filled only by a noded upon a termination of an sprocess.

disk use is the disk usage of a terminated sprocess. This field is filled only by a noded upon a termination of an sprocess.

23.2 Send a Message to User(s)

Usage: send a message to be displayed on (all) job representatives (e.g. "system going down in 5 minutes...").

Routing: There are a couple of possibilities:

1. Messages can be originated on masterd and sent to a single job representative to inform the user about the status of the job. See Section 19.1.
2. Messges can be sent from a master controller to masterd, and then be forwarded to all job representatives.
   Can only be sent by a superuser.
   See Section 20.2.

Structure:

0x0151 0x0451	Message text

24 The Job Loading Protocol in Detail

These messages are defined in `load.h`. They are used in the jr-md and nd-md protocols.

The way these messages interact with others is shown in Fig. 8. First, the job representative send a job submital message to masterd with the requested resources (jobsub, Section 27.1). The masterd allocates nodes, and sends all of them a job configuration update message (jobcfgupd, Section 26.4). At the same time, it sends an acknowledgement to the job

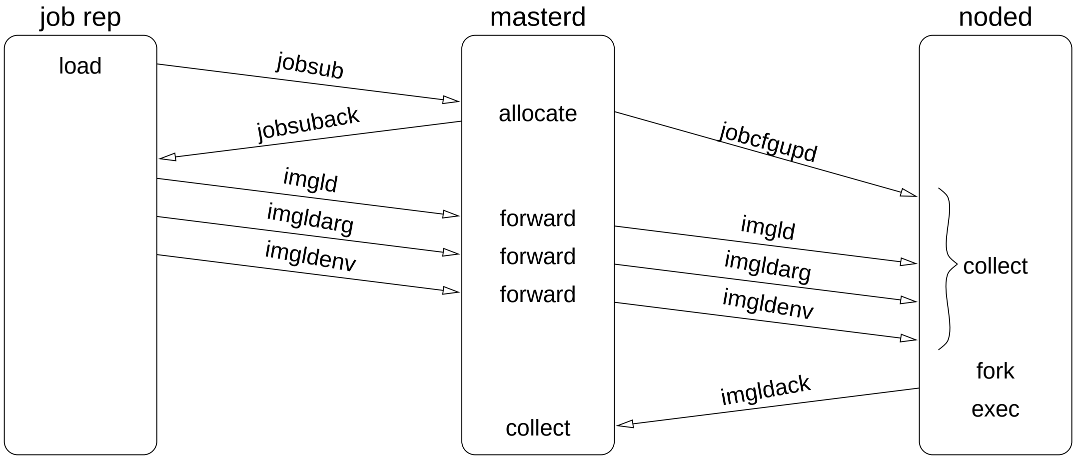


Figure 8: Messages involved in loading jobs.

representative (**jobsuback**, Section 27.2). Then the job representative sends a sequence of the three messages described here to actually convey the information needed to load the job: its path (**imgld**), arguments (**imgldarg**), and environment (**imgldenv**). These are forwarded by the masterd to the affected nodes. (actually, the job representative does not have to wait for the jobsuback: it may send the imgld messages immediately). Each noded then does a fork/exec to create its sprocess, and reports back to the masterd using an image load acknowledge message (described below, even though strictly speaking it belongs in md-nd). The masterd collects these acknowledgements, to keep track of the status of the job as a whole. In some cases, more than one imgldack and jobsuback may be sent, if the status changes.

24.1 Image Load Request

Usage: A transfer of information leading to loading and executing a job. Note that the image itself is not transferred via the masterd — it is obtained directly by the nodeds using NFS mounting.

See Section 11.

Routing: originally from the job representative to masterd, and then forwarded from masterd to noded.

Structure:

	4B	4B	1B	2B	2B	2B	2B	4B	
0x0214 0x0414	uid	xpid	slot	JR stdio port	JR stderr port	JR LED port	DBG port	IP address	···

...	path to executable image

Uid is the user ID for running this job.

Xpid is the xProcess ID for this job.

Slot is the slot to use in the noded's sproctab (for gang scheduling).

JR stdio port is the port to be used in order to establish standard input and standard output connections with the job representative.

JR stderr port is the port to be used in order to establish standard error connections with the job representative.

JR LED port is the port to be used in order to establish LED I/O connections with the job representative.

DBG port is the port to be used by the debugger back end to communicate with the debugger front end, if any. If no debugger control is expected, the DBG port is 0.

IP address is the job representative's IP address.

Path of the executable to run. This is a string with a predefined maximum length.

> *To be determined:* Need to figure out how to pass the $path shell variable so that the noded will be able to search for the executable in cases where a full path is not provided. Possible solution: the environment PATH variable.

Comments: this is the first of 3 messages that together constitute an image load request.

In addition, the job configuration information is sent separately using a Job Configuration Update message (see Section 26.4).

24.2 Arguments for Image Load Request

Usage: transfer the arguments to the executable.

See Section 11 and Section 19.1.

Routing: originally from the job representative to masterd, and then forwarded from masterd to noded.

Structure:

	4B	1B	
0x0215 0x0415	xpid	slot	arguments

Xpid is the xProcess ID for this job.

Slot is where information is stored in the noded's sproctab.

Arguments are the arguments to the program given as a string just as they were entered by the user. The nodeds later parse this string to construct the argv array for execve.

Comments: this is the second of 3 messages that together constitute an image load request.

24.3 Environment for Image Load Request

Usage: transfer the environment to the executable.

See Section 11 and Section 19.1.

Routing: originally from the job representative to masterd, and then forwarded from masterd to noded.

Structure:

	4B	1B	
0x0216 0x0416	xpid	slot	environment encoding

Xpid is the xProcess ID for this job.

Slot is where information is stored in the noded's sproctab.

Environment is the environment is encoded as a set of zero-terminated strings (i.e. a vector of vecors). Each one starts with an environment variable name, followed by an "=", and then the variable's value. Not all the environment is passed automatically — only the part specified by the user (see Section 19.1 and qualification in Section 11).

Comments: this is the third of 3 messages that together constitute an image load request.

24.4 Image Load Acknowledgement

Usage: acknowledge an image load.

See Section 11 and Section 24.

Routing: from noded to masterd.

Structure:

	4B	1B	4B	4B
0x0217	xpid	xpid ent	status	errno

Xpid identifies the job in question.

Xpidentry is where information about this job is stored in the masterd'd xpidtab.

Status is used to indicate success or failure, with the reason.

Errno is the specific Unix error code, if relevant.

Comments: this is sent only after all 3 load messages have been accepted and processed by the noded. There may be more than one Image Load Acknowledge messages sent to masterd: for example, a success ack may be sent after the fork, and then a fail message after the exec fails.

A successful parallel program initiation constitutes a successful Image Load Ack from each and every noded sent an Image Load Request.

25 The MC-Masterd Protocol in Detail

The Master Control GUI serves two functions — the user may use the Master Control GUI in order to request information on the state of the parallel machine as a whole, i.e., in order to examine the pool and view the current job configuration, as well as display system load and other sundry actions.

The super user may also use the MC in order to retrieve system and job logs, kill any running job, remove a node from the pool and, of course, shut the parallel machine down.

In addition to the messages described below, this protocol includes the following messages which are shared with other protocols:

metasignals , described in Section 23.1

show system message , described in Section 23.2

25.1 Pool Request

Usage: request information about the current system configuration.

See Section 20.1.

Routing: from master controller to masterd.

Structure:

0x0141

25.2 Pool Transfer

Usage: send current system configuration information.

See Section 20.1.

Routing: from masterd to the requesting master controller.

Structure:

	1B	1B	1B	4B	1B	1B	2B	4B	4B	
0x0142	Ent Cnt	SID	T	IP address	C	A	clk	memory	disk	More pool entries

Ent Cnt is the number of pool entries, which should match the system size.

For each one, the following information is given:

SID is an entry's system node ID.

T is an entry's node type (compute, I/O, etc.).

IP address is an entry's network address.

C is the number of CPUs in the node

A is an encoding of the architecture

Clk is the clock rate

Memory is an entry's memory capacity.

Disk is an entry's disk capacity.

25.3 Job Configuration Request

Usage: request information about the current system load.

See Section 20.1.

Routing: from master controller to masterd.

Structure:

	1B	4B
0x0143	type	xpid

type is general or detailed. General means a list of all jobs. Detailed means a list of the nodes used by a specific job.

xpid is the job for which detailed information is requested.

25.4 General Job Configuration Transfer

Usage: send load information, in the form of a list of jobs and their characteristics.

See Section 20.1.

Routing: from masterd to a master controller.

Structure:

	2B	4B	4B	1B	1B	2B	
0x0144	entry cnt	xpid	uid	sz	st	time	More job entries

Entry Cnt is the number of jobs in the system.

For each one, the following information is given:

xpid is the job ID.

uid is the user ID.

sz is the size (number of nodes used).

st is the status (waiting or running).

time is the time in this status (in seconds). With gang scheduling, this is runtime, not wall clock time.

25.5 Detailed Job Configuration Transfer

Usage: send detailed information regarding the nodes used by a job.

See Section 20.1.

Routing: from masterd to a master controller.

Structure:

	4B	2B	1B	1B	
0x0145	xpid	total nodes	SID	JID	More SID/JID pairs

xpid is the job's XPID

Totalnodes is the number of nodes used by the job.

SID is the system node ID.

JID is the job logical node ID.

25.6 Remove Node

Usage: removes a given node from the pool.

See Section 20.2.

Routing: from master controller to masterd.
Can only be sent by a superuser.

Structure:

	1B	4B
0x0161	SID	grace period

SID is the system node ID of the node to remove.

Grace Period is the time till sprocesses currently running on the node are killed. If 0, they are killed at once. If larger, they are given this grace period to exit, and only then the node is removed.

Comments: the masterd handles the grace period, and then sends a shutdown message to the node's noded.

26 The Masterd-Noded Protocol in Detail

The Masterd-Noded protocol carries most of the control load in ParPar. In addition to the messages described below, this protocol includes the following messages which are shared with other protocols:

load job , described in Section 24.1

arguments for job , described in Section 24.2

environment for job , described in Section 24.3

metasignals , described in Section 23.1

shutdown , described in Section 22.2

set logging level , described in Section 22.1

26.1 Capability Message

Usage: used by a noded to bind to masterd and inform it about the available resources on this node.

See Section 10.4.

Routing: from noded to masterd.

Structure:

	1B	1B	2B	4B	4B
0x0203	C	A	clk	Memory	Disk

C is the number of CPUs in the box (in case it is an SMP).

A is an encoding of the architecture.

Clk is the clock rate, in MHz.

Memory is the node's memory, expressed in kilobytes.

Disk is the node's disk capacity, expressed in kilobytes.

26.2 Capability Acknowledge

Usage: acknowledgemnt of the receipt of a capability message.

See Section 10.4.

Routing: from masterd to noded

Structure:

	1B
0x0204	SID

SID is the system's internal ID for this node, as assigned by the masterd.

26.3 System Configuration Update

Usage: inform nodes of changes in the current picture of the parallel machine. This can be used to incrementally add and delete nodes.

See Section 10.4.

Routing: from masterd to nodeds.

Structure:

	4B	1B	1B	2B	4B	
0x0241	Entry Count	SID	act	flags	IP address	More SID/act/flags/IP entries

Entry Count is the number of entries in the packet.

SID is the system internal node identifier.

act is the action: add or delete the node.

IP address is the node's network address.

Comments: this will most probably not be used.

26.4 Job Configuration Update

Usage: notify nodes of changes in the configuration of a job, i.e. added or deleted nodes. This is also used to notify them of the initial configuration when the job is first loaded.

See Section 11 and Section 24.

Routing: from masterd to nodeds.

Structure:

	4B	1B	1B	2B	1B	1B	
0x0213	xpid	xpid ent	slot	total nodes	SID	JID	More SID/JID pairs

Xpid is the job's XPID.

Xpidentry is the entry in the masterd's xpidtab used to store information about this job.

Slot is the slot allocated for this job in the noded's sproctab.

Totalnodes is the number of node entries in the message, which is also the number of nodes in the job's partition.

SID (System Node ID) is the node number in the parallel machine as a whole.

JID (Job Node ID) is the xpid internal node number allocated to the accompanying System Node ID.

Comments: job Configuration Updates during execution (to notify remaining nodes of the termination of sprocesses on other nodes) may or may not be used, depending on the implementation of the user-level communication.

The question of whether to include the network address in the job configuration, or send a separate system configuration, is still open (see Section 10.4).

26.5 Heartbeat

Usage: request for a heartbeat.

See Section 15.1.

Routing: from masterd to noded.

Structure:

0x0232

26.6 Heartbeat Acknowledgment

Usage: the heartbeat itself, showing that the node is alive.

See Section 15.1.

Routing: from noded to masterd.

Structure:

0x0233

27 The JR-Masterd Protocol in Detail

The Job Representative communicates with the master daemon only when jobs are initiated and terminated. In addition to the messages described below, this protocol includes the following messages which are shared with other protocols:

load job , described in Section 24.1

arguments for job , described in Section 24.2

environment for job , described in Section 24.3

metasignals , described in Section 23.1

show system message , described in Section 23.2

27.1 Job Submittal

Job submittal is mediated by a sequence of packets. The first is a request for the allocation of nodes. The rest are a description of the job, and are forwarded by the masterd to the allocated nodeds. Only the initial request is described here. The others may be found in Sections 24.1, 24.2, and 24.3

Usage: request an allocation of nodes in order to run a job.

See Section 11 and Section 24.

Routing: from job representative to masterd.

Structure:

	1B	4B	4B	
0x0411	B	memory	time	node requirement

B is a bitfield with the following bits:

Batch if set, run in batch mode, i.e. the job may be queued and there is no connection with the job representative

Debug if set, run under debugger control

Verbose if set, give detailed messages about everything

Memory the total required memory for the job (sum on all nodes, in KB)

Time a bound on how long the jub should run, provided it is given its maximal request of processors

Node requirement is the number of nodes requested, as described in Section 19.2. The format is binary: each 4 bytes are an integer, which represents one optional size. If a range is specified by the user, this is translated to a list of all the sizes in the range. Furthermore, the numbers should be sorted in increasing order.

Comments: "node allocation" does not necessarily mean that they are dedicated. With gang scheduling, they will not be dedicated.

27.2 Submittal Acknowledge

Usage: inform about a status change for a submitted job.

See Section 11 and Section 24.

Routing: from masterd to the job representative.

Structure:

	4B	4B
0x0412	status	nodes

status is success or the reason for a failure, if any:

- Unauthorized.
- Not enough resources.
- Temporarily not enough resources; request queued.
- Fork/exec failure on nodes.

nodes is the number of nodes actually allocated. It is meaningful in cases where the user specified a range of options.

Comments: more than one such message may be sent, e.g. initially the job may be queued, and then when it is actually loaded the executable may not be found.

28 The JR-Sprocess Protocol in Detail

In order to decrease the load on masterd and noded's, job representatives and sprocesses will transfer standard I/O and LED requests directly from one to another. Standard I/O is done by simply redirecting the stdin, stdout, and stderr file descriptors to sockets, so there is no protocol involved. Only LEDs require messages to be defined.

The tagging of each packet with the xpid is retained for logging and debugging.

28.1 Query or Request LEDs

Usage: query the number of LEDs available for each node.

See Section 14.3 and Section 19.1.

Routing: from sprocess to job representative.

Structure:

0x0872

28.2 Define Number of LED's

Usage: define or supply the number of LEDs available for each node.

See Section 14.3 and Section 19.1.

Routing: there are two uses:

1. From sprocess to job representative, in which case this is a request to define the number of LEDs as directed.
2. From job representative to sprocess, as an answer to a LED query message.

Structure:

	1B
0x0871	N

N is the number of LED's defined for each sprocess.

Comments: Any number of leds may be defined (within reason). These LED's are numbered consecutively, starting from 0.

28.3 Turn LED On

Usage: turn a LED on, i.e. give it a ceratain color.

See Section 14.3 and Section 19.1.

Routing: from sprocess to job representative.

Structure:

	1B	1B
0x0873	N	C

N is the LED number.

C is the new LED color.

Comments: a color spectrum needs to be defined.

Part VI
Additional Services

29 Logging

The ParPar system shall maintain multiple levels and types of logs. The first distinction is between the accounting log and the activity logs: the accounting log lists all jobs submitted and the resources they used (maintained only on the host), while the activity logs list everything that happens on any node in the system. Thus the activity logs can be classified as the master log and the node logs. In addition, the contents of the activity logs pertain both to system activity and to user jobs, so an additional classification is into system entries and job entries in the log. Each job entry is tagged with its xpid.

29.1 Accounting Log

The accounting log contains the following information about each job submitted to the system:

1. User's uid
2. Executable run
3. Nodes allocated
4. Submit date and time
5. Load date and time
6. Termination date and time (last sprocess)
7. Runtime (with gang scheduling, this is different from the difference between load and termination)
8. Termination status (success, fail, killed, removed before loaded)
9. List of resource usage on each node, including
   (a) CPU usage, both user and system
   (b) Memory usage
   (c) Disk usage
   (d) Termination date and time (for this sprocess)

The goal is not so much to charge users, as to learn about the workload [12]. The information about the resources used by each sprocess is sent to masterd as part of the message informing it that the sprocess has terminated. This message is a meta-signal (see Section 23.1.

29.2 Activity Logs

Logs are created both on the master and on the nodes and may be amalgamated automagically on request. This is done by interleaving the different logs based on timestamps.

> *To be determined:* We need to figure out how to do the merging correctly. For this, the clocks on the different nodes should be as closely synchronized as possible. In addition, it is wise to incorporate clock messages in the log itself to maintain data about the differences remaining between the clocks, as in UTE [26].

Activity logging entails logging the protocol packets at the lowest level to logging system startup information on the highest level. It is subdivided into the following levels of detail. Each level implies all higher levels.

LOG_CRIT Critical error which **prevents the reporting component from continued execution.** The component will exit with an error status immediately.

Eg., failure to open socket for listening, failure to select() existing open sockets, failure to locate critical configuration files or resources, etc.

LOG_ERR Error which **prevents completion of current action** but does not harm the reporting component otherwise.

Eg., broken connection, failure to allocate non-critical resources (gui table, node table), inconsistent data in internal tables, etc.

LOG_WARNING Unexpected condition encountered.

Eg., spurious messages incoming from a connection, potential (and detected) buffer oversteps, etc.

LOG_NOTICE Config changes.

Eg., New component has connected with the reporting component, a component has been deregistered, etc.

LOG_INFO Component and subcomponent activation (complete actions).

Eg., Successful job load, etc.

LOG_DEBUG Everything, including each function entered, and in some cases, packet contents.

The default level is LOG_NOTICE. Other levels may be requested by a command line option, or via the master controller interface.

29.3 Mechanics of Logging

There are a number of options regarding how to keep the logs:

1. Logs can be maintained in a flat ASCII file, kept flushed at all times.
2. Have two levels of logs: The primary log is a circular buffer of an as yet undetermined size, while the secondary log is kept in a file.
3. Use Unix logging facilities via the `syslog` system call. This is probably the best approach in terms of our headaches, but might have performance implications, because the actual logging is done be a daemon process (syslogd).

Initially, we shall use Unix logging. To do so, the (Unix) system has to be configured as described in Section 33. The above description of levels is based on the syslogd definitions. We shall use one of the LOCAL facilities to distinguish our log from those of other system services — LOCAL0 for master daemon, LOCAL1 for node daemon.

30 Debugging

The ParPar debugger will be an extension of a conventional debugger divided into two modules: a front end that will deal with multiple nodes by focusing on single nodes or subsets of nodes, and a back end that will control the individual sprocesses. The conventional debugger we will use will probably be `ddd`, which is itself a graphical front-end for `gdb`.

30.1 User Interface

> *To be determined:* The following is a new design from scratch. An alternative is to use the `ddd` GUI with some extensions. Also, we need to consider the option of having multiple windows open at once, focused on disjoint sets of nodes; how will this be coordinated?

The front end will have a graphical user interface, similar to that of the job representative, but with some changes. An example is given in Fig. 9, and includes the following features:

- A display of the program text, with marks for breakpoints and the current location of various processes. If all processes are at the same place, a larger arrow will be used; otherwise, there will be a number of small arrows.
- A node status map, in which each node is represented by a small colored square. A possible color code is

  red — at breakpoint

  yellow — waiting for some event, e.g. a message arrival

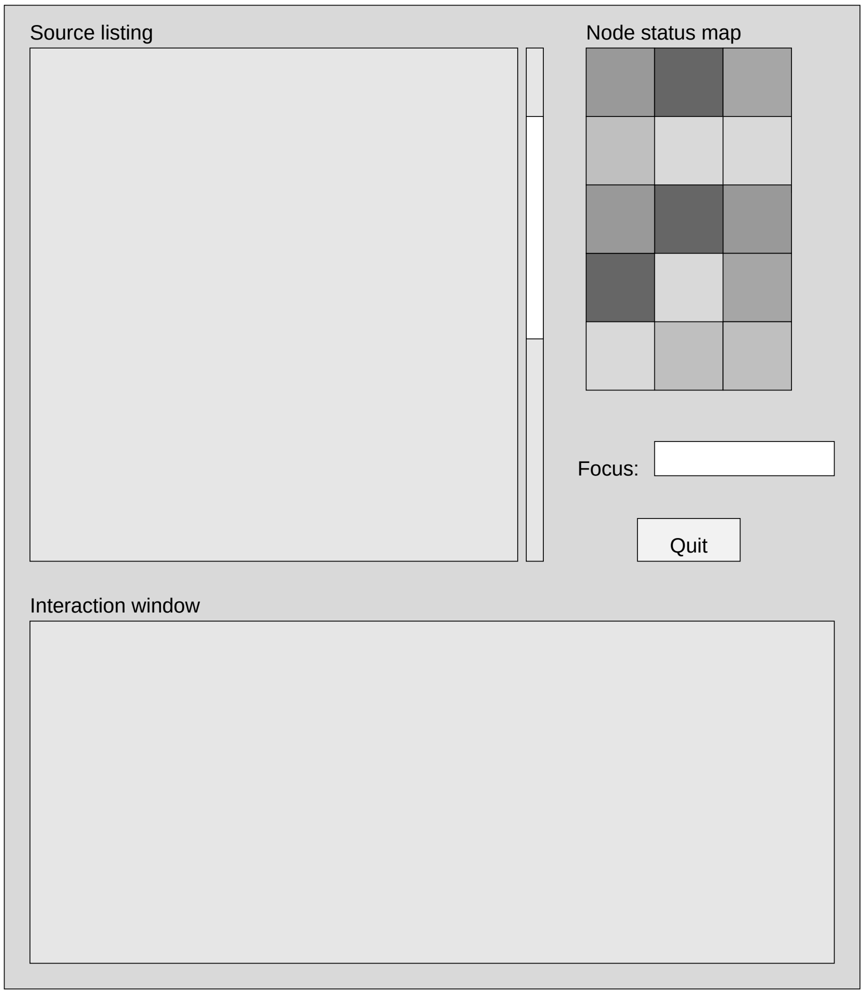


Figure 9: Sketch of parallel debugger interface.

blue — stopped somewhere (while single-stepping)

green — running

black — terminated

Squares may also display a numerical code, e.g. the breakpoint's number.

- Indication of focus, i.e. which processes are currently targeted. It is possible to change the focus by mousing on one of the squares in the display (to focus on a single process), or by mousing on an arrow in the listing (to focus on all the processes at that location).
- An interaction window, in which conventional debugger commands (like print variable, dump stack, etc.) can be sent to the focused processes, and the results are displayed.

30.2 Control Structure

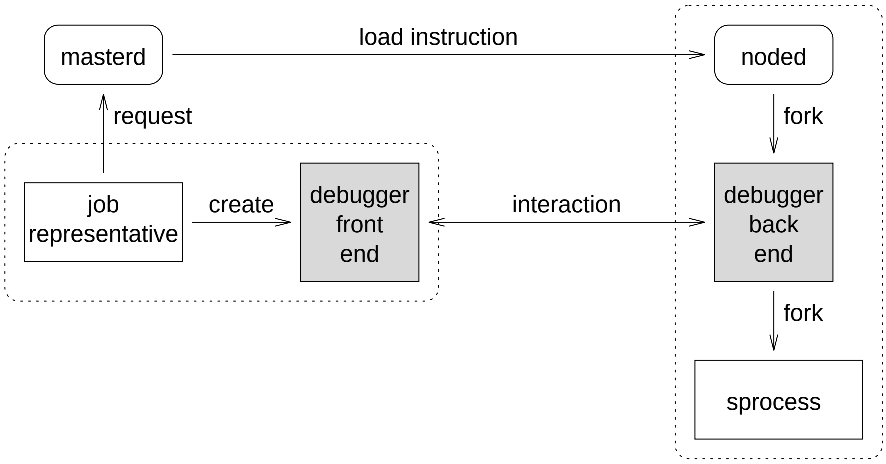


Figure 10: Control structure for debugging.

To start up a job under debugger control, the job representative will indicate that debugging is desired when the job is submitted. The node daemons will then start a debugger on each node, rather than starting the sprocesses directly. These debuggers will establish communication links with the front-end debugger that will be started in the meanwhile by the job representative. This structure is show schematically in Fig. 10.

> *To be determined:* Still missing: debugging from `core` files. The problem is that all the sProcesses run in the same user directory, so if they core-dump, the core files will overwrite each other. Can we find a way to generate separate core files? Note that the core files *cannot* be generated in the local disk on each node, because there is no guarantee that the same nodes will be re-allocated for the debugging run. Actually, when debugging core files we don't need the parallel machine at all — just direct the debugger at the (multiple) core files.

31 Implementation of Gang Scheduling

packing and allocation [9]

decision about multi-context switching: either broadcast each time from the master daemon, or table-based if we can synchronize clocks well enough.

implementation of multi-context switching: coordination of the node daemons, use of Unix signals to stop and start the relevant sprocesses (as done at LLNL [15]).

32 Implementation of User-Level Communication

low-level design - probably port the U-Net implementation for fast Ethernet [25]. this requires a kernel patch to allocate protected memory.

implementation of MPI and PVM interfaces, probably also by porting them from public domain implementations.

Part VII
Appendixes

33 Installation

In order to install the ParPar software on a cluster of Unix systems, the following modifications have to be made to system files:

- /etc/services: add

```
parjr        10000/tcp
parnd        10001/tcp
parmc        10002/tcp
parfail      10003/tcp
```

- /etc/syslog.conf: add

```
local0.*     /var/log/masterd.log
local1.*     /var/log/noded.log
```

  This has to be at the beginning of the file.

34 Open Issues and Questions

This is a list of generalized questions pertaining to the parallel project.

- The BIG question: is BSDI the right choice? In particular, should we go for Linux instead (i.e. keep the Unix direction, but use a different version)?

	BSDI	Linux
availability	free including source	free including source
support	company + system group	user community
stability	relatively stable	somewhat volatile
software	little	used by many others
modification	???	???

- Is our set of software components rich enough, or are we loading too much on each module? for example,

  - should there be another level of indirection between the noded and the sprocess? that is, should each sprocess have its own watchdog process that takes care of it, rather than having noded care for all sprocesses?
  - should masterd be divided into a number of separate processes for control, resource management, etc?

- Is an active resource manager (masterd) better than a passive one (a set of scripts executed directly by the job reps via rsh and accessing a shared database of resources)?

- The whole issue of interaction between messaging on the control network and local scheduling of the noded and other processes needs to be clarified. In particular, what exactly happens when as incoming messages causes an interrupt?

- The information needed for user-level communication has to be determined. Does each node have to keep and updated map of all nodes participating in each job, requiring job configuration update messages when something changes, or can we do away with that?

35 Terminology

Binding: The process by which a parallel computer is established by the various daemons running on the nodes.

The binding process has two goals: first, to establish a communication regime between masterd and the nodeds, and second, to obtain capability information for the purpose of allocation of primacy in certain tasks in a heterogeneous parallel environment.

Capability: Information regarding the resources available to a node. The initial design does not use this capability but for future expansion, in a heterogeneous hardware environment, given nodes may be favourably "discriminated" for certain tasks, depending on the capability information.

Control Network: Network used to transfer control messages among system components.

Data Network: Network dedicated to user-level communication among the compute nodes. This network has better characteristics than the control network.

Extended Process (xprocess): The total sum of interdependent processes running on more than one node and the amalgamation of which forms a single parallel job.

The xprocess on a given parallel machine has a process ID — XPID — which is unique to the total parallel machine.

Gang Scheduling: Simultaneous context switching across a set of nodes.

Host Node: Synonym for the master.

I/O Node: A node used (exclusively) to support a parallel file system.

Job: Synonym for xProcess.

Job Representative (JR): A GUI tool for initiating, controlling and following the output of parallel programs.

LED: A status indicator in the Job Representative. An xprocess may define any (within reason) number of LEDs, which are associated with its sprocesses. The sprocesses can change the status (color) of the LEDs at will, as a simple form of graphical output.

Load: The action of starting an xprocess running on a set of nodes, by passing to them information about the executable and the configuration.

Master Node (front-end, host): A single or multiple CPU machine presenting a unified interface to the total parallel machine before a regular user. The master runs the master daemon, masterd.

Masterd: A daemon running on the master node (either outside or, in effect, inside the kernel), responsible for configuration management, resource management, and signaling at the xprocesses level.

Master Control: A user interface to masterd used for system level and administrative interaction.

Meta Signal: A signal sent between the system daemons (noded and masterd), pertaining to a whole xprocess, and conveying information about signals at the sprocess level.

Signals originating with the master are broadcast or multicast by the master to the nodes, while signals originating with a node are "reflected" from the master — the signal is sent to the master which, subsequently, broadcasts or multicasts it, if the need exits.

Node (back end node, compute node): A node is a single or multiple CPU machine participating in a parallel machine. The node is not independent and is, in fact, invisible to the regular user. Nodes are connected to the master by a control network, and to each other by a dedicated high-speed network. They run the node daemon, noded.

Noded: A daemon running on a node, responsible for transmitting status and signal information from the node to the master and for receiving signals from the master.

Pool: A list or table of nodes, listing nodes in the parallel computer, their network identifications (IP addresses), capability information, and other data (node types — for a heterogeneous environment — defining gross node capabilities) collected by the master and used to assign nodes to xprocesses and vice versa.

Process: An independent process running on a single node, servicing and being serviced by that node only. A process is NOT a member of an extended process.

Slot: In the context of gang scheduling, resources are two dimensional: one dimension is the pool of nodes, and the other is slots of scheduling time. The sprocesses of an xprocess are allocated to the same slot across multiple nodes, so that they will be scheduled in unison.

Sub Process (sprocess): A single process participating in an xprocess and running on a single node. An sprocess cannot be and is not independent.

An sprocess on a given parallel machine has a process id — SPID — which is unique to the node on which it executes but may or may not be unique to the total parallel machine.

Subpool: A subset of the pool, transmitted to the nodes. The subpool includes only the node number, its network address and the node type.

(node) Type: Either the host, a compute node, or an I/O node (for future implementation of internal parallel I/O).

User-Level Communication: Communication across the data network without operating system (i.e. Unix) intervention.

Xprocess (xProc): Short for "extended process".

36 Coding Standards

- Code must be readable by a child of two. Keep it simple and straight forward. In all likelihood, the code will have to be understood and modified by someone else.
- Use indents of 4 spaces.
- Same-line opening brace, e.g.

```
if (a < b) {
    a = 0;
    b = 1;
}
```

- Use a _t suffix for typedefs.
- Use UPPERCASE for macros/defines.
- Use a pp_ prefix for global variables.
- Use a _pp_ prefix for internal library functions.
- Use the following function definitions style:

```
/*
 * Comment block
 */
PRIVATE/EXTERNAL type
function(type var1, type var2)
{
    ...
}
```

- External variable declration in module.h:

```
EXTERN type var;

#ifdef EXTERNS_HERE
  #define EXTERN
#else
  #define EXTERN extern
#endif
```

- Protection of .h files:

```
#ifndef FILE_H
#define FILE_H
...
#endif
```

- Include RCS identification in all files

```
#ifndef lint
static char RCSid[] = "$Id: code.tex,v 1.1 1997/05/15 11:47:54 feit Exp $
#endif
```

In .h files, use the file name instead of "id" in the variable name.

References

[1] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team, "*A case for NOW (networks of workstations)*". *IEEE Micro* **15(1)**, pp. 54–64, Feb 1995.

[2] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph, "*ParC—an extension of C for shared memory parallel processing*". *Software — Pract. & Exp.* **26(5)**, pp. 581–612, May 1996.

[3] M. 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] S. H. Bokhari, "*The Linux operating system*". *Computer* **28(8)**, pp. 74–79, Aug 1995.

[5] J. B. Chen, Y. Endo, K. Chan, D. Mazières, A. Dias, M. Seltzer, and M. D. Smith, "*The measured performance of personal computer operating systems*". *ACM Trans. Comput. Syst.* **14(1)**, pp. 3–40, Feb 1996.

[6] D. Citron, D. G. Feitelson, and I. Exman, "*Parallel activity roadmaps*". In *Parallel Computing: Trends and Applications*, G. R. Joubert, D. Trystram, F. J. Peters, and D. J. Evans (eds.), pp. 593–596, Elsevier Science B.V., 1994.

[7] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "*A message passing standard for MPP and workstations*". *Comm. ACM* **39(7)**, pp. 84–90, Jul 1996.

[8] D. G. Feitelson, "*Deadlock detection without wait-for graphs*". *Parallel Comput.* **17(12)**, pp. 1377–1383, Dec 1991.

[9] 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. Lecture Notes in Computer Science Vol. 1162.

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

[11] D. G. Feitelson, Y. Ben-Asher, M. Ben Ezra, I. Exman, L. Picherski, L. Rudolph, and D. Zernik, "*Issues in run-time support for tightly-coupled parallel processing*". In 3rd *Symp. Experiences with Distributed & Multiprocessor Syst.*, pp. 27–42, USENIX, Mar 1992.

[12] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[13] Y. I. Friedman, D. G. Feitelson, and I. Exman, "*The parallel break construct, or how to kill an activity tree*". In 10th *Intl. Parallel Processing Symp.*, pp. 230–234, Apr 1996.

[14] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 User's Guide and Reference Manual.* Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.

[15] B. Gorda and R. Wolski, "*Time sharing massively parallel machines*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 214–217, Aug 1995.

[16] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[17] M. J. Litzkow, M. Livny, and M. W. Mutka, "*Condor - a hunter of idle workstations*". In 8th *Intl. Conf. Distributed Comput. Syst.*, pp. 104–111, Jun 1988.

[18] MPI Forum, "*MPI: a message-passing interface standard*". *Intl. J. Supercomput. Applications & High Performance Comput.* **8(3/4)**, Fall/Winter 1994.

[19] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[20] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson, "*Robust partitioning schemes of multiprocessor systems*". *Performance Evaluation* **19(2-3)**, pp. 141–165, Mar 1994.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "*Design and implementation of the Sun network filesystem*". In *Proc. Summer USENIX Technical Conf.*, pp. 119–130, Jun 1985.

[22] K. Shteiman, D. Feitelson, L. Rudolph, and I. Exman, "*Envelopes in adaptive local queues for MIMD load balancing*". In *Parallel Processing: CONPAR 92 – VAPP V*, pp. 479–484, Springer-Verlag, Sep 1992. Lecture Notes in Computer Science vol. 634.

[23] J. Skovira, W. Chan, H. Zhou, and D. Lifka, "*The EASY - LoadLeveler API project*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 41–47, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[24] V. S. Sunderam, "*PVM: a framework for parallel distributed computing*". *Concurrency — Pract. & Exp.* **2(4)**, pp. 315–339, Dec 1990.

[25] 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.

[26] C. E. Wu, H. Franke, and Y-H. Liu, "*A unified trace environment for IBM SP systems*". *IEEE Parallel & Distributed Technology* **4(2)**, pp. 89–93, Summer 1996.