Monitors, Messages, and Clusters: the p4 Parallel Programming System

Ralph M. Butler *
College of Computing Sciences and Engineering
University of North Florida
Jacksonville, FL 32224
rbutler@sinkhole.unf.edu

Ewing L. Lusk †
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
lusk@mcs.anl.gov

Abstract

p4 is a portable library of C and Fortran subroutines for programming parallel computers. It is the current version of a system that has been in use since 1984. It includes features for explicit parallel programming of shared-memory machines, distributed-memory machines (including heterogeneous networks of workstations), and clusters, by which we mean shared-memory multiprocessors communicating via message passing. We discuss here the design goals, history, and system architecture of p4 and describe briefly a diverse collection of applications that have demonstrated the utility of p4.

1 Introduction

p4 is a library of routines designed to express a wide variety of parallel algorithms portably, efficiently and simply. The goal of portability requires it to use widely accepted *models* of computation rather than specific vendor implementations of those models. The goal of efficiency requires it to use models of computation relatively close to those provided by the machines themselves and their system software. And the goal of simplicity requires it to provide programmers with a relatively small number of concepts, while providing a rich enough set that they can express the algorithms they have designed.

These goals are not always consistent. In some cases, the inconsistency has been resolved in p4 by providing multiple ways to do things. (For example, p4 provides completely automatic buffer management, but if a programmer prefers to deal with it himself to avoid the overhead of an extra copy operation, then p4 provides the appropriate buffer-management routines.) In other cases, judgments have been made regarding the balancing of portability, efficiency, and simplicity considerations. In many situations, considerable complexity has been absorbed into p4 itself in order to provide simplicity and portability to the programmer.

*This work was partially supported by National Science Foundation grant CCR-9121875.

†This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under contract W-31-109-Eng-38.

The most distinguishing feature of `p4` is its support for multiple models of parallel computation. For the shared-memory MIMD model, it provides the *monitor* paradigm [14] for coordinating access to shared data, and runs on "true" shared-memory computers such as the Sequent symmetry and Alliant FX/2800, as well as NUMA (non-uniform memory access) machines that provide a shared-memory computational model, like the BBN TC-2000 and Kendall Square KSR-1. For the distributed-memory MIMD model, it provides the "usual" typed *message-passing* functions and global operations, and supplies implementations on all the platforms that support this model, such as the Intel Touchstone Delta and TMC CM-5, shared-memory machines such as the Sequent Symmetry and Kendall Square KSR-1, and heterogeneous networks of workstations. It also provides for explicit management of *clusters*, in which both shared- and distributed-memory MIMD models are explicitly used at the same time. It provides no support for the SIMD computational model.

In the following sections, we describe `p4` in detail. Section 2 outlines the history of that branch of portable parallel programming research at Argonne that has given rise to `p4`, and explains its relationship with other systems. Section 3 outlines the basic functions in the library and describes some more advanced features as well. Section 5 describes the implementation, including some interesting aspects of `p4` not visible to the user. Section 6 describes a representative sample of `p4` applications that illustrates some of the uses to which `p4` has been put by its users. Finally, we mention some related projects and enhancements, largely done by others, that have added useful features to `p4`, and conclude with some reflections and future plans.

2 Background

`p4` is a third-generation system. Here we trace its background (see Figure 1) in order to show which constructs have held up and which have changed as the system has evolved.

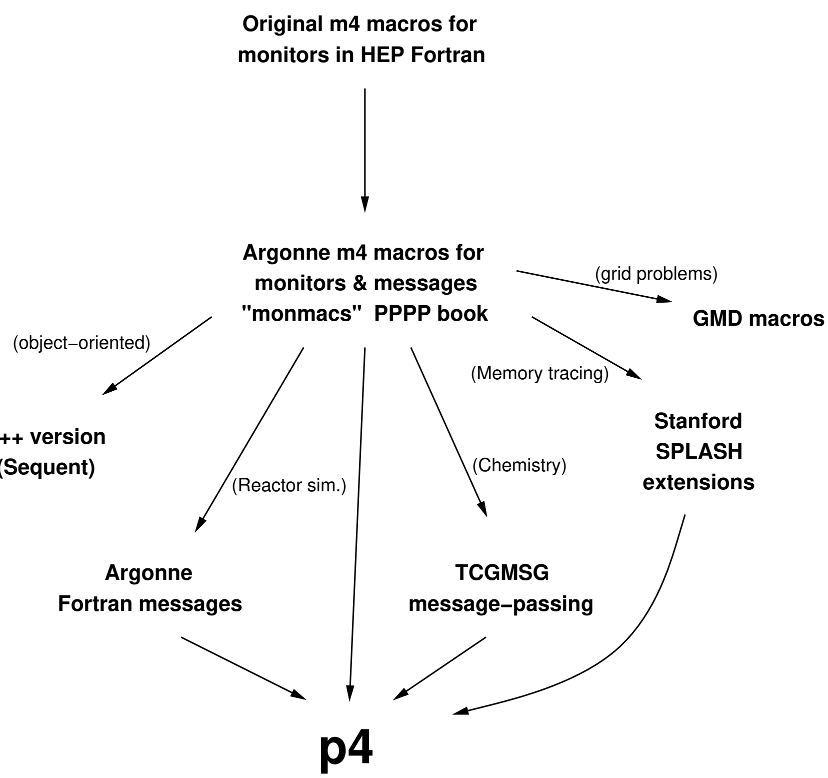

Figure 1: `p4` Family Tree

2.1 The Beginning

In 1984 Argonne National Laboratory acquired a Denelcor HEP (Heterogeneous Element Processor) [15], the first commercial multiprocessor and the first machine in what would become Argonne's Advanced Computing Research Facility (ACRF). The HEP was a true shared-memory machine with a multistage pipeline that made it appear to have between 8 and 12 processors. (A larger version was available at the Army Ballistics Research Laboratory.) Argonne scientists quickly came to grips with the problem of how to program this machine.

The HEP was the first in a long line of parallel computers that have been delivered with familiar sequential languages (the HEP had only Fortran) but unfamiliar, nonportable, and proprietary extensions for accessing and controlling parallelism. In the case of the HEP, the extension consisted of "asynchronous variables" by which annotations on individual Fortran variables triggered synchronizations in the hardware, leading to a type of dataflow synchronization in Fortran programs. The mechanism was efficient, possibly even elegant, but extremely difficult to program with. As a result, each group of researchers at Argonne that used the HEP moved quickly from dealing with HEP Fortran extensions to creating a programming environment they could use for applications.

Ewing Lusk and Ross Overbeek chose *monitors* as their central paradigm for controlling access to shared data by multiple processors [21]. A monitor is an abstract data type encapsulating shared data, initialization instructions, and critical code sections. Monitors were extensively studied for their theoretical properties in the early seventies in the context of a scientific approach to operating systems [11, 14]. We have found them a durable construct for shared-memory programming.

To implement monitors on the Fortran-only HEP, we defined macros for monitor operations and used the `m4` macro processor to expand them on our VAX into HEP Fortran, and then transferred the code to the HEP for compilation [20]. This system worked well, and we were able to use the HEP quite productively [7, 17].

2.2 Monmacs

As parallel processing entered the commercial marketplace, the ACRF at Argonne expanded in 1985 to include machines from Alliant, Encore, Sequent, and Intel. The HEP macros for monitors were ported to the Alliant, Encore, and Sequent and were expanded to include C as well as Fortran. The package was still `m4`-based, but compilation was simpler because `m4` ran on the machines themselves. The package did not have an official name at this point, but because of its macro definition basis and the focus on monitors it was sometimes called "monmacs" or "parmacs." The package was expanded to include message passing for all three of the environments that supported the distributed-memory model: the iPSC/1, the (new) workstation network, and the shared-memory machines. Thus the original motivation for the invention of the library, namely, to be able to program with a well-understood programming paradigm, gave way to portability as the main motivation. It was at the end of this period that those who had collaborated on the implementation wrote the book *Portable Programs for Parallel Processors* [3], which served to publicize the system widely. The code itself was distributed with the book and also over the network. The ACRF staff held regular classes in parallel computing, and used "`monmacs`" to teach students how to program on many parallel machines at the same time.

2.3 The Interim Period

During the next few years (1987–1989), development of the system by its originators tapered off, and the gap thus created was filled by others who built a variety of interesting and useful systems that either used parts of its code or were inspired by `p4`. The version included in the book [3] had only rudimentary support for message passing in Fortran, so that was added by Dave Liebfritz at

Argonne. Bob Beck at Sequent did a C++ version of the monitors part [1]. Rolf Hempel at GMD greatly improved the Fortran interface and added extensive functionality on top of it for grid-based computation. He borrowed the word PARMACS, and what is now called PARMACS is his system, widely used in Europe [2]. Robert Harrison of Argonne's Chemistry Division re-implemented the message-passing subsystem to provide more efficiency, better error handling, and a library of global operations in his system TCGMSG [12]. Many of his enhancements were later incorporated into `p4`. The SPLASH group at Stanford built their collection of shared-memory applications [24] on `monmacs`, adding instrumentation macros to support assorted research projects in parallel programming.

2.4 The Current System

In 1989 Ralph Butler and Ewing Lusk began a complete rewrite of the entire system, with the goal of producing a very robust system for wide distribution that included features present from the beginning as well as new features (message passing among heterogeneous machines, global operations) that users were requesting. There were also a great variety of new parallel machines and workstations to support. This effort, concentrated in 1990 and 1991, produced the current `p4`, which takes its name from the title of the 1987 book [3] and its functionality from all previous versions, going all the way back to the HEP. The interface has changed little since `p4` was first released, although many performance improvements have been made, and programs written for the `p4` of 1989 run essentially unchanged on machines only released in 1993.. While a number of features have been added, little has been removed, and virtually all `p4` programs ever written will run on the current version (which, as this is being written in April of 1993, is version 1.3).

3 General Capabilities

In this section we describe the basic user interface for the various computational models supported by `p4`. We focus on the C interface; Fortran-callable versions of most routines are also available. The C interface contains more sophisticated functions and provides the programmer more opportunities for optimization.

3.1 The p4 Process Model and Starting Up

`p4` tries to be as flexible as possible about how processes will be started. It is assumed that the user executes a command in response to a system prompt. Depending on the particular environment, this may start a large number of processes or may start a single process, which in turn will start others. The precise disposition of processes (the machines they will run on, the executable files, and the grouping into clusters of processes that share memory) is usually specified in a "process group file," or "procgroup file" for short. The name of the procgroup file is typically a command-line argument to a `p4` application, and the function `p4_create_procgroup`, called by the initial or "master" process, reads the file and starts up the collection of `p4` processes. Since a user may wish to obtain the information normally specified in the procgroup file in his own way, access is also provided to the data structures normally filled in by `p4_create_procgroup`; in this case `p4_startup` is called to start the processes based on these data structures. On machines where all processes are started externally at the same time, the `p4_create_procgroup` performs necessary initialization, after which the "master" process can join in the general computation, as in the example in Section 3.7. Thus `p4` can be used to program in the "SPMD" model (single program, multiple data) as well as in the "master-slave" model.

All `p4` programs must execute a `p4_initenv` call before any other `p4` calls; in particular, a process does not have a `p4` process id—an integer between 0 (the initial process) and one less than the total number of processes—until it has executed `p4_initenv`. Then `p4_get_my_id` returns a process's own `p4` identifier.

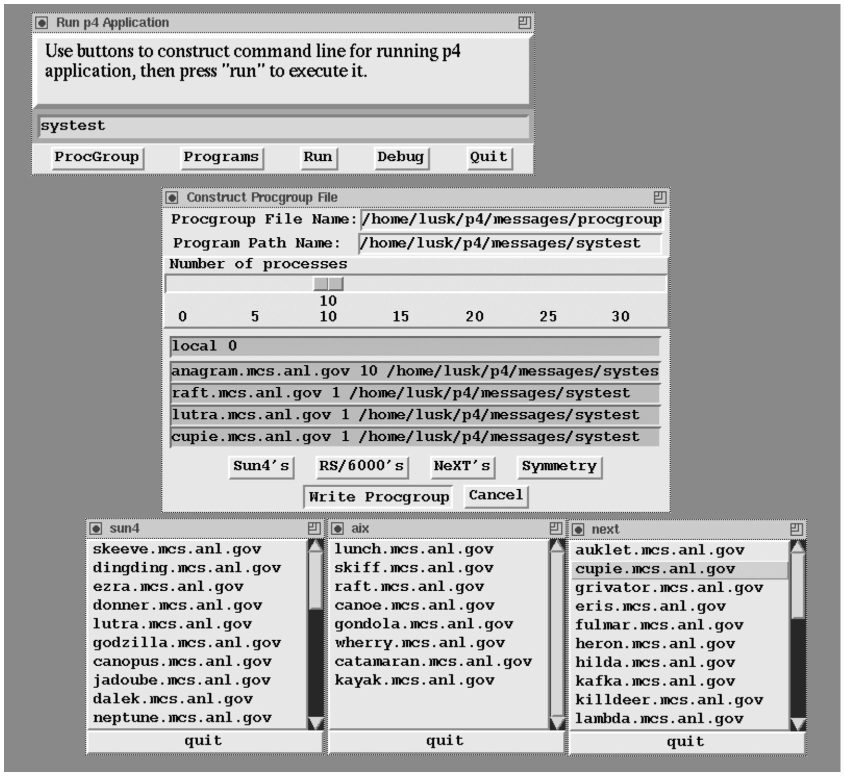

Figure 2: Graphical interface to `p4`

Recently an experimental graphical user interface has been constructed to make it easier to run `p4` programs. It is shown in Figure 2. It dynamically locates workstations on the network and makes it easy to construct a valid procgroup file and then run a `p4` application with just a few mouse clicks. This system was easily built using Tcl/Tk [23].

3.2 Shared Memory and Monitors

The most primitive components of a shared-memory computational model are semaphores and locks, and these are what the vendor libraries typically supply. One can argue that programming with these concepts is somewhat like programming with branch instructions. Just as the "if-then-else" and "do-while" of structured programming can be thought of as a structured use of branch instructions (which, after all, are still there in the compiler-generated machine code), so monitors can be thought of as a structured use of locks.

The `p4` system provides the *monitor* data type in C by type definitions, and also a collection of useful monitors. The data encapsulated in monitors resides in shared memory, which is managed with the `p4_shmalloc` and `p4_shfree` functions. The lack of memory-management functions in Fortran makes it difficult to describe monitors in a portable way, and so we dropped support for monitors in Fortran in `p4`, at least temporarily. They may resurface eventually (see Section 8). The fundamental monitor operations (see [14]) are provided by `p4_menter`, `p4_mexit`, `p4_delay`, and `p4_continue`. With these, users can define their own monitors in a completely portable way. Over the years we have seldom found it necessary to write new monitors, finding it quite adequate to rely on a small set of monitors defined in the earliest versions of `p4`. These include `p4_barrier`, to synchronize all or a subset of the processes, `p4_getsub` for expressing loop-level parallelism, and `p4_lock` and `p4_unlock` for occasional low-level operations. The most useful monitor of all has been what we call the "askfor" monitor, whose fundamental operation is `p4_askfor`, which returns a user-defined task. The `p4_askfor` and its related routines `p4_update`, `p4_probend`, and `p4_progend`, provide sufficient functionality that a user may write a customized, general dispatching algorithm for a particular application. The `p4` User's Guide [4] provides details and examples.

Note that `p4` does not implement monitors on distributed-memory machines, since then the computational model would be too far away from the hardware for the efficiency that `p4` aims for. The monitor operations are only provided on platforms that supply to the programmer a shared-memory computational model.

3.3 Distributed Memory and Messages

`p4` provides facilities for sending and receiving typed messages by `p4_send` and `p4_recv`. These are "blocking" operations in the sense that when the `p4_send` returns, the buffer may be reused, and when the `p4_recv` returns, the message is in the buffer. The "probe" operation (`p4_msgs_available`) allows one to test for the arrival of a message before committing to wait for it. Optional versions of `p4_send` specify synchronous operation (`p4_sendr` does not return until the message has been received by the execution of a `p4_recv` at the destination), heterogeneous communication (`p4_sendx` has a data type argument that prescribes translation into standard data format with `xdr`), or user-managed buffers (`p4_sendb` requires that a buffer be preallocated by `p4_msg_alloc` to avoid the overhead of copying data from the user's memory into a formatted `p4` buffer).

3.4 Clusters

Management of a collection of processes, some subsets of which share memory, utilizes both the monitor and message-passing parts of the `p4` library. In addition, there are routines for identifying a "cluster master" process in each cluster (`p4_am_i_cluster_master`), finding out the number of clusters (`p4_num_cluster_ids`), and finding out the process identifiers of the processes in one's own cluster (`p4_get_cluster_ids`). These routines and other support the explicit expression of algorithms that take advantage of the cluster environment.

3.5 Miscellaneous Functions

A number of `p4` library functions are of general utility. Program sections can be timed to the millisecond with the `p4_clock` routine, and on most machines to the microsecond with the `p4_usclock` routine. The function `p4_dprintf` can be used to print user messages, which `p4` will route back to the terminal where the `p4` application was started. The message will be automatically tagged with the identifier of the process that issued it. `p4_dprintfl` takes a level number as an argument, so that messages can be filtered by debugging level, which is a command-line argument. Since `p4` itself is instrumented with `p4_dprintfl`'s (although a compile-time switch can remove them), it is possible to get detailed traces of the internal activities of `p4`, sometimes necessary to catch even a user bug. `p4_error`, called by either `p4` or the user, attempts to produce a useful error message and then causes all processes to exit.

3.6 Collective Operations

`p4` provides a number of collective operations, particularly useful in the distributed-memory model. `p4_global_op` takes as one of its arguments an operation to be performed on distributed data. It is thus relatively easy to expand the set of collective operations. Currently supplied are maximum, minimum, absolute-value-maximum, absolute-value-minimum, sum, and product. Each operates on a vector of integers, floats, or doubles.

3.7 An Example p4 Program

For a flavor of what a `p4` program looks like, we give in Figure 3 example of a program in which all processes send messages to one another. The program here is in C, but the Fortran would look quite similar.

```
#include "p4.h"
#define  GREETING  100                    /* message type for greeting   */

main(argc,argv)
int argc;
char **argv;
{
    p4_initenv(&argc,argv);               /* initialize self             */
    if (p4_get_my_id() == 0)              /* if first process            */
        p4_create_procgroup();            /*     initialize others       */
    worker();                             /* all processes do same work  */
    p4_wait_for_end();                    /* wait for orderly shutdown   */
}

worker()                                  /* all processes do this       */
{
    char *incoming, *msg = "hello";
    int myid, size, nprocs, from, i, type;

    myid = p4_get_my_id();                /* who am I?                   */
    nprocs = p4_num_total_ids();          /* how many all together?      */
    for (i=0; i < nprocs; i++)
    {
        if (i != myid)
            p4_send(GREETING, i, msg, strlen(msg)+1);   /* send msg      */
    }
    for (i=0; i < nprocs - 1; i++)
    {
        type = from = -1;                 /* receive any message         */
        incoming = NULL;                  /* automatic buffer allocation */
        p4_recv(&type,&from,&incoming,&size);           /* recv msg      */
        p4_dprintf("%d received msg=:%s: from %d\n",myid,incoming,from);
        p4_msg_free(incoming);            /* free buffer                 */
    }
}
```

Figure 3: A Simple `p4` Program

To run this program on either an Intel iPSC860 or a Sequent Symmetry, one would start it with the following procgroup file:

```
machine.anl.gov  32  /home/usr/p4progs/machine/example
```

To run the same program on a network of workstations, one would just change the procgroup file to something like:

```
local          0
sun1.anl.gov    1    /home/usr/p4progs/sun/example
sun2.anl.gov    1    /home/usr/p4progs/sun/example
ibm1.anl.gov    1    /home/usr/p4progs/ibm/example
ibm1.anl.gov    1    /home/usr/p4progs/ibm/example
```

4 Performance

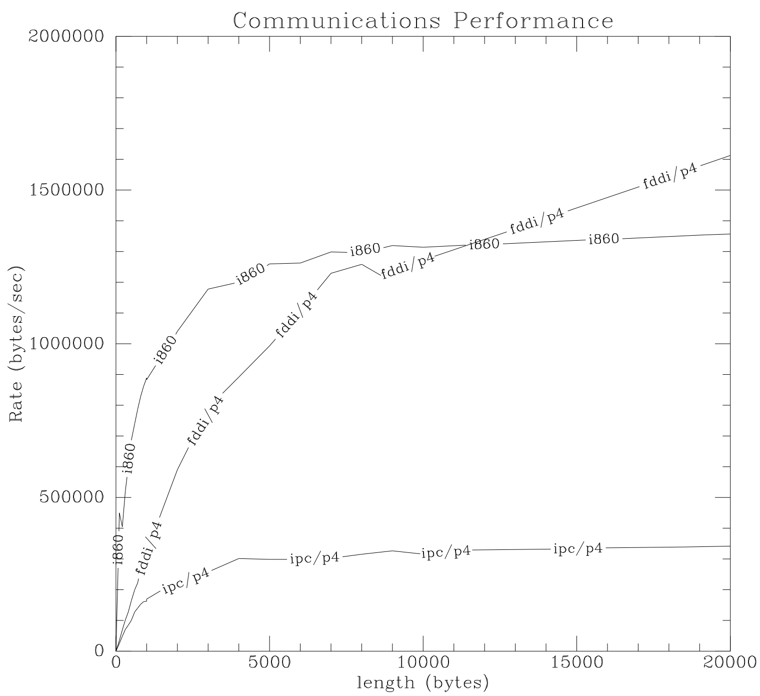

Figure 4: **p4** Network Performance

Efficiency is a major goal of **p4**, and has been the focus of considerable effort. Although there is still room for further optimization, **p4** does not currently add a significant amount of overhead to the underlying transport layer. Although applications are more important than benchmarks, we have occasionally tested **p4** in "benchmark" mode to make sure that it is reasonably efficient. In Figure 4 we compare the *round-trip* transmission rates between two processes for a number of message sizes. The curve marked "ipc/p4" represents **p4** on an ethernet connecting two Sparc 10 workstations. The curve marked "fddi/p4" shows that **p4** is fully able to take advantage of a faster transmission layer and that an FDDI-connected workstation network can compete with a specialized message-passing machine. The curve marked "i860" shows message-passing performance of the same program converted to run with Intel primitives. (The barely visible "zigzag" at 128 bytes shows that the test program is sensitive enough to pick up the protocol change that occurs on the iPSC/860 at that message length.)

5 Architecture

In this section we describe some noteworthy features of the current `p4` implementation.

5.1 Starting Processes

Normally the user starts one process, which uses the information in the procgroup file to start other processes. For each remote cluster, the process uses the Unix remote shell command `rsh` to start one of the processes, called the "cluster master" on the specified machine. Then each cluster master creates the processes it will share memory with via the Unix `fork` call. Processes can also be created explicitly by `p4_create`, which forks one process that will share memory with its parent.

An alternative to the remote shell mechanism is provided for faster startup, particularly on workstation networks. In many environments a "server" process can be started on the remote machines ahead of time. When `p4` wants to start a process on a remote machine, it first attempts to contact the server. If the server is present, then it forks the remote process, which happens quickly; if the server is not running, then the usual remote shell mechanism is used. A system like this must be constructed quite carefully if it is not to compromise the security of the network. The server distributed with `p4` takes all of the precautions of `rsh` and then some. It requires preregistration of the application in a file readable only by the user, and may require a password from the user in the middle of the startup process. The server method is required when remote shell commands are not permitted on the network. Scripts are supplied with `p4` to manage a collection of servers.

5.2 Monitors

Shared memory is managed by `p4_shmalloc` and `p4_shfree`, which have different implementations on different shared-memory machines. On those with very primitive memory management systems for shared memory (for example, no "free" operation), we have implemented a simple and portable memory management subsystem inside of `p4`. We are thus able to present the same interface to the programmer no matter what the underlying system is.

The most fundamental operation for shared-memory programming is the *lock*. We use spin locks on most systems, although a few machines have more sophisticated locking functions that we use when possible. It is then possible to layer the basic monitor-building primitives (`p4_enter`, etc.) on top of the locks, and then the library of monitors (`p4_barrier`, `p4_askfor`, etc.) on top of these. Thus almost all of the code in the implementation is completely portable; to implement monitors on a new shared-memory machine, one needs only a new set of definitions for locks. For example, the current implementation of `p4` does not take advantage of special vendor-specific barrier code, although it would be relatively easy to extend the system to do so.

5.3 Message Passing

A small amount of performance has been sacrificed in order to make most of `p4`'s implementation code portable. During a `p4_send`, the user's data is copied into a `p4` buffer, which contains a 40-byte header. (This step is bypassed if the user obtains the buffer complete with header by means of `p4_msg_alloc` and builds his message in it.) Once the buffer has been packed with the message and the header information (destination, sender, message type, length, data type, acknowledge-request-flag), `p4` looks up the destination in a table to determine how to deliver it. If the sending and receiving process share memory, then the buffer is just placed in the destination process's queue. If there is a machine-specific send operation available (e.g., the two processes are on an iPSC/860), then the appropriate vendor-specific send operation is used. If the message must travel over a

TCP/IP network, then if a socket is already open to the destination process, it is used; otherwise a socket is opened first. Note that only connections that are going to actually be used are opened. Currently such sockets are left open, and one can run out of them (although this seldom happens; modern workstations support lots of open sockets). A more sophisticated `p4` implementation may close sockets to reuse their file descriptors. `xdr` is used to translate messages between machines with different data formats. This is done only when absolutely necessary; `p4` contains a table of those pairs of machines that require translation.

In some cases the copying of messages into `p4`-maintained buffers can speed things up. On the Intel iPSC/860 and DELTA we can use Intel's `isend` and return immediately to the user instead of `csend`, which blocks until the message has been sent. The `p4` buffer is flagged as "in use," and the `isend` is `wait`ed on only when the buffer is really needed later, by which time the `isend` will probably have completed.

During a receive operation, all possible sources of incoming messages are checked until the criteria (source and type) specified on the `p4_recv` are satisfied. For transmission layers where the size of a message is made available before the message is read (this is done on TCP/IP networks by reading the header before the rest of the message), a buffer is allocated for the message to be read into. The `p4_recv` returns a pointer to this buffer. Thus a user need not know ahead of time the size of the message. Alternatively, the user can allocate a buffer ahead of time. This approach allows reuse of the same area by the user for multiple messages.

Allocation and deallocation of buffers are optimized by maintaining a pool of available buffers of varying sizes. The sizes of these buffer pools can be set by the user, using `p4_set_buf`. The default is to maintain pools for messages of sizes 64, 256, 1K, 4K, 16K, 256K, and 1M.

5.4 Clusters

When one is combining the shared- and distributed-memory models, some subtle issues arise regarding local and global synchronization. For example, the cluster slaves that are forked off by the cluster master need to wait until it has obtained shared memory, which is under user control, before commencing operations on shared memory. They can't wait in an ordinary barrier because a user barrier is created in user-obtained shared memory, which is exactly what they don't have at the time they are created. Therefore a special built-in barrier is provided for this purpose, accessed by `p4_cluster_mem_synch`.

One unusual implementation of `p4` was on the short-lived Alliant CAMPUS system, which had HiPPI switches connecting shared-memory Alliants. For this machine we and Robert Harrison wrote an interface to the HiPPI switch that would support `p4` and TCGMSG types of operations. It is exactly this type of hardware configuration that `p4`'s cluster model is for.

6 Experiences

`p4` has hundreds of users around the world. We describe here a sampling of applications that demonstrate the wide range of ways in which `p4` has been used.

Monitors: Automated Reasoning. Argonne has long been in the forefront of automated reasoning research. The parallelization of our primary theorem prover presented a particular problem because of its central shared data structures. The tight interweaving of indices and shared substructures makes this an intrinsically shared-memory application. We used `p4`'s shared-memory operations, in particular the `p4_askfor` monitor, to implement the algorithm presented in [25] and got excellent results [19], even for the very fine-grained parallelism necessary for this application. We developed it on a Sequent Symmetry and ran it for peak performance on an Alliant FX/2800.

Portable Message-Passing: Phylogenetic Trees. `p4` was used to develop a parallel version of the "maximum likelihood" method of computing phylogenetic trees from RNA sequence data [22]. The largest such tree ever computed was derived using `p4` and the Intel Touchstone Delta. The portability of the program enabled it to be developed on a Sequent (which has a good debugging environment), run on small data sets on workstation networks, tested further on the Intel iPSC/860, and finally run in production mode on the Intel Delta, all without changing a single line of source code. Most recently, the same code has been used to discover new phylogenetic trees on the IBM SP-1.

Clusters: Full Configuration Interaction. Another application comes from the area of computational chemistry. This application fits best into the shared-memory model, in that a global data structure is both read and updated in random patterns by all of the processes. Using `p4`'s cluster model, we ported this program to a collection of four 25-processor Alliant FX/2800's connected by a HiPPI switch. That is, processes on a single cluster (one 25-processor Alliant) used monitors to coordinate access to their part of the shared data and to schedule themselves (using `p4_askfor`). Data in other clusters was accessed by sending messages to a data-manager process in that cluster. Again, `p4`'s portability made it possible to develop the program on a single Sequent Symmetry (procgroup options allow the partitioning of a single shared-memory machine into several that communicate via messages) before running it on a set of Alliants connected by the HiPPI switch.

Heterogeneous Computing: Piezoelectric Crystals. A large finite-element code for computing resonances in piezoelectric crystals was written by Mark Jones and Paul Plassmann in conjunction with Motorola [6]. The primary computation was done on the Delta, and `p4` was used for communication with the file server (a large-memory Solbourne located 2000 miles away), and a Stardent Titan for displaying graphical output.

Invisible Usage: Superconductivity. An example of usage of `p4` without the user "knowing it" is the work reported in [8]. Here, the users are modeling vortex dynamics in high-temperature superconductors using the three-dimensional, time-dependent Ginzburg-Landau equation as a phenomenological model. The data movement between processors is done by using the the BlockComm [9] system. The development and debugging of this code were done on Sun workstations and then moved into production on the Intel DELTA.

Large Workstation Networks: Test Pattern Generation. Peter Krauss at the Technical University of Munich has been using `p4` to do test pattern generation [16] on a homogeneous network of Hewlett-Packard workstations. He has run his `p4` application on as many as 102 workstations at the same time. He also runs the same program on an Intel iPSC/860 and on a heterogeneous network made up of a Sequent Symmetry together with workstations from DEC, HP, and Sun.

Portable Application Benchmarks. A number of groups are beginning to assemble collections of "real" application programs that can be used to measure performance on parallel machines. Two of these such efforts are the SPLASH project at Stanford [24] and the Perfect Club Benchmarks. Portable versions of the codes make them much more useful. The SPLASH codes have been converted to use `p4`'s shared-memory operations, and work is in progress to convert the Perfect Club benchmarks to `p4`'s message-passing operations.

7 Related Systems

`p4` is designed to be used as part of other systems and is currently playing a role in several projects.

p4-Linda. An implementation of the Linda programming model has been done, using both shared-memory and distributed-memory models for the the underlying hardware [5].

BlockComm and Chameleon. `p4` is one of the "machines" on which Bill Gropp's `Chameleon` system [10] runs. Thus it is possible to run a program that has been coded for the Intel iPSC/860, say, and run it on a workstation network using `p4`, without changing source code. Chameleon is the foundation for the `BlockComm` communication library [9], which allows users to avoid explicit construction of messages when the messages consist of matrix subblocks.

MPI. Many vendors, users, and authors of message-passing systems have organized to try to define a standard message-passing interface. This "standard" is (as of this writing) still in flux, but parts of it are solidifying. Bill Gropp and Ewing Lusk are providing a reference implementation for the standard as it is developed, and this implementation uses the Chameleon system, and thus either `p4` or PVM, for its network implementation.

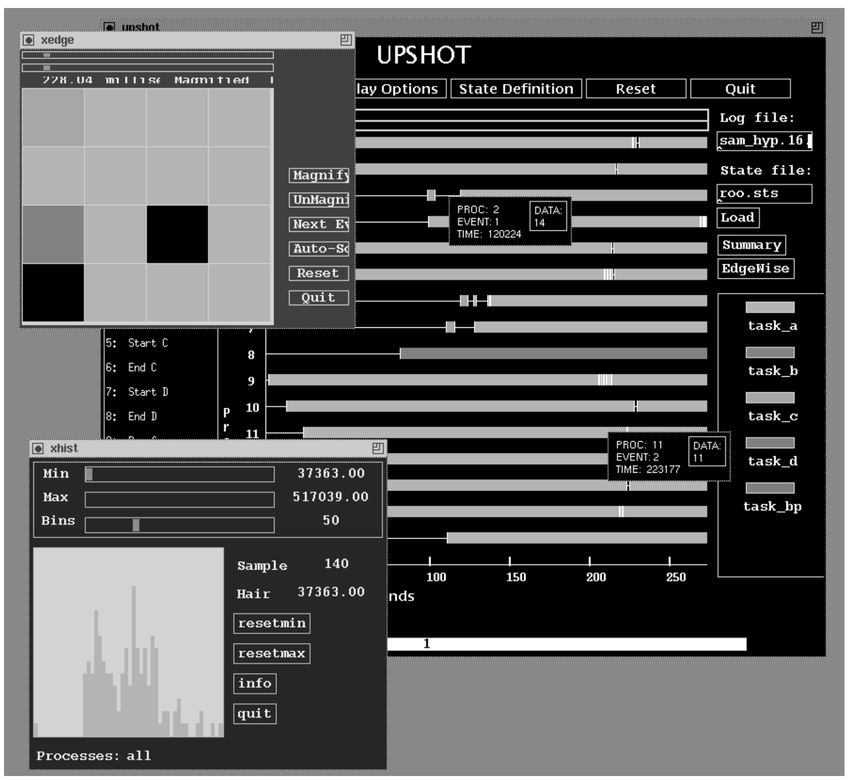

Figure 5: `upshot` Looking at Parallel Automated Reasoning Program

Parallel program visualization. Distributed with `p4` is a portable library called `alog` for producing log files of user-specified events. These files can then be examined with a variety of tools. One that we have used extensively for some time for studying the behavior of `p4` programs is `upshot`

[13, 18]. In Figure 5 we see `upshot` being used to display details of a run made on the parallel automated reasoning system described above. `alog` is also used to instrument `p4` internally. If certain compile-time switches are set, any `p4` program will produce `alog` logfiles. `alog` is not directly tied to `p4`; it has also been used to instrument a variety of other parallel programming systems.

Program animation. In Figure 6 we demonstrate a more elaborate program visualization system called PADL (Program Animation Display Language), which is currently under development. Here we see a graphical display of a `p4` program with a dynamically changing display of individual messages and accumulated statistics.

DQS. One difficulty with efficiently using a network of workstations as a parallel computer is scheduling a queue of parallel jobs. Jim Patterson at Boeing has developed a version of DQS that schedules `p4` jobs. This is being incorporated into the standard DQS distribution by Tom Green at Florida State.

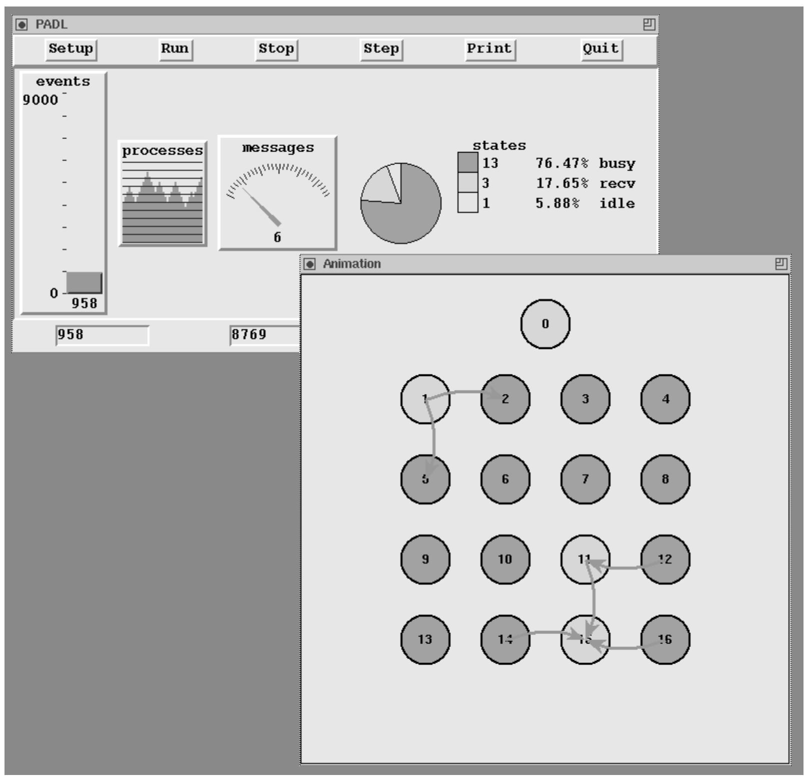

Figure 6: Examining logfiles with PADL

8 Status and Future Work

`p4` will continue to be a portable parallel programming system that incorporates new ideas in parallel computing. The next generation of operating system seems likely to offer a "thread" model for parallel programming that is still relatively unexplored. For shared-memory machines, monitors can play a useful role in providing a higher layer of operations than that supplied by thread packages themselves. They are also a useful layer at which to provide portability among various thread libraries, and for providing a shared-memroy programming model in Fortran. In the message-passing area, we hope that vendor implementations of MPI will replace `p4`'s portability layer with corresponding improvements in efficiency. And the cluster model will become more visible as it arrives in the form of multiprocessor workstations on networks (Sun, SGI), hierarchical machines (Convex), and multiprocessor nodes on multicomputers (Intel).

9 Acknowledgments

We thank Ross Overbeek, co-author of the two predecessor systems to `p4`. We thank especially our users, whose comments, complaints, and requests helped to design `p4` and make it sturdy. Particular thanks to go to local users Dave Levine, Barry Smith, and Bill Gropp, who performed the service and paid the price of being the first to use each new release.

10 Availability

`p4` is in the public domain and can be obtained by anonymous `ftp` to `info.mcs.anl.gov` at Argonne National Laboratory. It is also available through `netlib`. The distribution contains all the source code, a meta-makefile to build `p4` on any of the machines described below, a set of examples, and a User's Guide [4], which can be installed as an on-line help system as well, via the Gnu Emacs `info` mechanism. It is available for the following set of machines: Sequent Symmetry; Encore Multimax; Alliant FX/8, FX/800, FX/2800, and Campus; Cray X/MP and C-90; Sun, NeXT, HP,DEC, Silicon Graphics, and IBM RS/6000 workstations; Stardent Titan; BBN GP-1000 and TC-2000; Intel iPSC/860, Touchstone Delta, CM-5, and Paragon; nCUBE; KSR; and IBM SP-1. It is not difficult to port to new systems, and we intend to do these ports when new machines become available.

References

[1] Bob Beck. Shared-memory parallel programming in C++ (rational parmacs). In *Proceedings of the 2nd Annual Meeting, Sequent User's Resource Forum*, pages 187–205, 1988.

[2] L. Bomans, D. Roose, and R. Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.

[3] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, 1987.

[4] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

[5] Ralph M. Butler, Alan L. Leveton, and Ewing Lusk. p4-Linda: A portable implementation of Linda. In C. S. Rajhavendra and Salim Hariri, editors, *Proceedings of the Second International Symposium on High-Performance Distributed Computing*. IEEE Computer Society Press, 1993. (to appear).

[6] Tom Canfield, Mark Jones, Paul Plassmann, and Michael Tang. Thermal effects on the frequency response of piezoelectric crystals. In *New Methods in Transient Analysis, PVP-Vol. 246 and AMD-Vol. 143*, pages 103–108, New York, 1992. ASME.

[7] John Gabriel, Tim Lindholm, E. L. Lusk, and R. A. Overbeek. Logic programming on the HEP. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 367–411. MIT Press, 1985.

[8] N. Galbreath, W. Gropp, D. Gunter, D. Levine, and G. Leaf. Parallel solution of the three-dimensional, time-dependent Ginzburg-Landau equation. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, 1993.

[9] William Gropp. Blockcomm for fortran. Technical report, Argonne National Laboratory, 1993. (to appear).

[10] William Gropp and Barry Smith. Parallel programming tools user's manual. Technical report, Argonne National Laboratory, 1993. (to appear).

[11] Per Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice-Hall, Inc., 1977.

[12] R. J. Harrison. Portable tools and applications for parallel computers. *Intern. J. Quantum Chem.*, 40(847), 1991.

[13] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with `upshot`. Technical Report ANL–91/15, Argonne National Laboratory, 1991.

[14] C. A. R. Hoare. Monitors: An operating system structuring concept. *Comunications of the ACM*, pages 549–557, October 1974.

[15] Janusz S. Kowalik, editor. *Parallel MIMD Computation: The HEP Supercomputer and Its Applications.* Scientific Computation Series. MIT Press, 1985.

[16] Peter A. Krauss and Kurt J. Antreich. Application of fault parallelism to the automatic test pattern generation for sequential circuits, 1993. (to appear in Springer LNCS).

[17] E. L. Lusk and R. A. Overbeek. Use of monitors in FORTRAN: a tutorial on the barrier, self-scheduling DO-loop, and askfor monitors. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 367–411. MIT Press, 1985.

[18] Ewing L. Lusk. Visualizing parallel program behavior. In Adrian Tentner, editor, *High Performance Computing 1993: Grand Challenges in Computer Simulation*, pages 209–213. The Society for Computer Simulation Simulation, 1993.

[19] Ewing L. Lusk and William W. McCune. Experiments with ROO, a parallel automated deduction system. In B. Fronhoefer and G. Wrightson, editors, *Parallelization in Inference Systems (Springer Lecture Notes in Artificial Intelligence 590)*, pages 139–162. Springer-Verlag, 1992.

[20] Ewing L. Lusk and Ross A. Overbeek. Implementation of monitors with macros: A programming aid for the HEP and other parallel processors. Technical Report ANL-83-97, Argonne National Laboratory, December 1983.

[21] Ewing L. Lusk and Ross A. Overbeek. A minimalist approach to portable, parallel programming. In Leah H. Jamieson, Dennis B. Gannon, and Robert J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 351–362. MIT Press, 1987.

[22] Gary Olsen, Carl Woese, Ray Hagstrom, Hideo Matsuda, and Ross Overbeek. Inference of phylogenetic trees using maximum likelihood. In *Proceedings of the First Intel Delta Applications Workshop*, pages 247–262, 1992.

[23] John Osterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the Winter 1991 USENIX Conference*, pages 105–115. USENIX Association, January 1991.

[24] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992. Also Stanford University Technical Report No. CSL-TR-92-526, June 1992.

[25] J. Slaney and E. Lusk. Parallelizing the closure computation in automated deduction. In M. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, Vol. 449*, pages 28–39, New York, July 1990. Springer-Verlag.