
Design of Digital's Parallel Software Environment

Edward G. Benson
David C.P. LaFrance-Linden
Richard A. Warren
Santa Wiryaman

Digital's Parallel Software Environment was designed to support the development and execution of scalable parallel applications on clusters (farms) of distributed- and shared-memory Alpha processors running the Digital UNIX operating system. PSE supports the parallel execution of High Performance Fortran applications with message-passing libraries that meet the low-latency and high-bandwidth communication requirements of efficient parallel computing. It provides system management tools to create clusters for distributed parallel processing and development tools to debug and profile HPF programs. An extended version of dbx allows HPF-distributed arrays to be viewed, and a parallel profiler supports both program counter and interval sampling. PSE also supplies generic facilities required by other parallel languages and systems.

Digital's Parallel Software Environment (PSE) was designed to support the development and execution of scalable parallel applications on clusters (farms) of distributed- and shared-memory Alpha processors running the Digital UNIX operating system. PSE version 1.0 supports the High Performance Fortran (HPF) language; it also supplies generic facilities required by other parallel languages and systems. PSE provides tools to define a cluster of processors and to manage distributed parallel execution. It also contains development tools for debugging and profiling parallel HPF programs. PSE supports optimized message passing over multiple interconnect types, including fiber distributed data interface (FDDI), asynchronous transfer mode (ATM), and shared memory.¹

In this paper, we present an overview of PSE version 1.0 and explain why it was designed and selected for use with HPF programs. We then discuss cluster definition and management, describe the PSE application model, and discuss PSE's message-passing communication options, including an optimized transport for message passing. We conclude with our performance results.

Overview of PSE

Many researchers and computer industry experts believe that to achieve cost-effective scalable parallel processing, systems must be built using off-the-shelf components and not specialized CPUs and interconnects.^{2,3} In accordance with this view, we have designed Digital's PSE to support the building of a consistent yet flexible and easy-to-use parallel-processing environment across a networked collection of Alpha Generation workstations, servers, and symmetric multiprocessors (SMPs). Layered on top of the Digital UNIX operating system, PSE provides the system software and tools needed to group collections of machines for parallel processing and to manage transparently the distribution and running of parallel applications. PSE is implemented as a set of run-time libraries and utilities and a daemon process.

PSE version 1.0 is designed to support clusters consisting of 1 to 256 machines interconnected with any networking fabric that Digital UNIX supports with the

transmission control protocol/internet protocol (TCP/IP). Networking technologies can range from simple Ethernet to FDDI, ATM, and MEMORY CHANNEL. Parallel execution is most efficient when the interconnect technology offers high-bandwidth and low-latency communications to the user at the process level. When building a cluster for parallel processing, the bisectional bandwidth of the communications fabric should scale with the number of processors in the cluster. In practice, such a configuration can be achieved by building clusters using Alpha processors and Digital's GIGASwitch/FDDI as components in a multistage switch configuration.^{4,5} Figures 1 and 2 show two examples of PSE cluster configurations. Although the design center for PSE is a set of machines connected by a high-speed local area interconnect, a cluster can be constructed that includes remote machines connected by a wide area network.

PSE is a collection of many interrelated entities that support parallel processing. PSE's model is to collect machines (called *members*) into a set (called a *cluster*). The members are generally all the machines at a site or within an organization that have or might have PSE installed. One then subsets the cluster into named (*partitions*) that may overlap. The members of a partition usually share some common attribute, which could be administrative (e.g., the machines of the development group), geographic (e.g., connected to the same FDDI switch), or relevant to the configuration (e.g., large memory, SMP).

The members of a cluster, the partitions, and other related data form a configuration database that can be maintained in different ways, but preferably by a system administrator. The configuration database can be distributed using the Domain Name System (DNS) or as a simple file distributed by Network File System (NFS).⁶ A daemon process *farmd* runs on each member to provide per-member dynamic information,

such as availability and system load average. The static database plus the dynamic information allow applications to perform tasks such as load balancing.

HPF Program Support

PSE was designed to be largely language-independent; it currently supports the HPF programming language. HPF allows programmers to express data parallel computations easily using Fortran 90 array-operation syntax. As a result, users can obtain the benefits of parallel processing without becoming systems programmers and developing message passing or threads-based programs. The HPF language and compiler are discussed elsewhere in this issue of the *Digital Technical Journal*.⁷

Writing parallel applications in HPF is significantly less complex than decomposing a problem and coding a solution using explicit message passing, but good development tools are required. To allow the viewing of HPF distributed arrays, we developed an extended version of *dbx* and a parallel profiler that supports both program counter and interval sampling. These tools are discussed later in this paper.

High performance and efficient communication are essential to success in parallel processing. PSE includes a private message-passing library for use with compiler-generated code. Thus it avoids overhead such as buffer alignment and size checking that are required with user-visible programming interfaces, such as Parallel Virtual Machine (PVM).⁸ The message-passing library supports shared memory and both TCP/IP and user datagram protocol (UDP)/IP protocols on many types of media, including FDDI and ATM. PSE also includes an optional subset implementation of the UDP, known as *UDP_prime*, that has been optimized to reduce latency and improve efficiency. This optimization is discussed later in this paper.

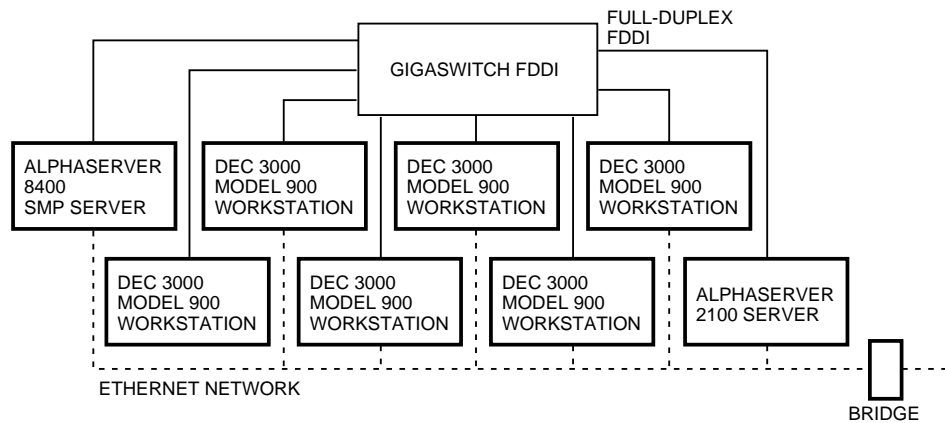


Figure 1
PSE Basic Configuration

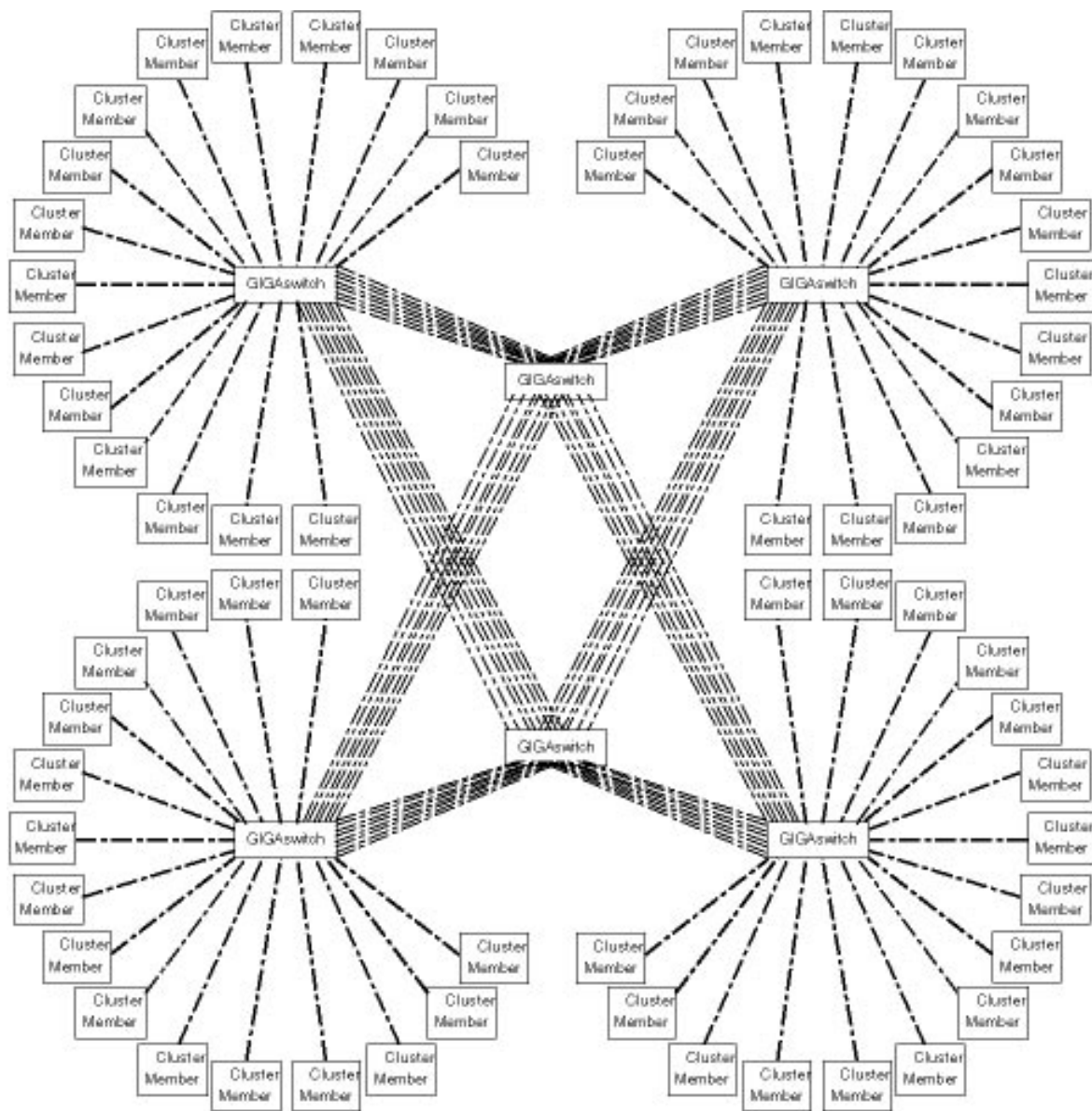


Figure 2
PSE Multistage Switch Configuration

Before developing PSE for use with HPF programs, Digital considered two major alternatives: the distributed computing environment (DCE) and PVM.^{8,9} (At that time, the message-passing interface [MPI] standard effort was in progress.¹⁰)

Although a good model for client-server application deployment, DCE is designed for use with remote CPU resources via procedure calls to libraries. This model is very different from the data-parallel and message-passing nature of distributed parallel processing. Its synchronous procedure call model requires the extensive use of threads. In addition, DCE contains a significant number of setup and management tasks. For these reasons, we rejected the DCE environment.

Three major considerations in our choice to develop PSE instead of using PVM were stability, performance, and transparency. At the start of the PSE project, the publicly available version of PVM did not meet the stability, performance, and transparency goals of the PSE project.

Cluster Definition and Management

PSE is designed to operate in a common system environment where systems are organized so that user access, file name space, host names, and so on are consistent. The ultimate goal for the systems in a distributed parallel-processing environment is to approach

the transparent usability of a symmetric multiprocessor. Facilities such as NFS (to mount/share file systems among machines, in particular working directories) and network information service (NIS) (also known as “yellow pages” and used to share password files) are frequently used to set up a common system environment. In such an environment, users can log into any machine and see the same environment. Other distributed environments such as Load Sharing Facility (LSF) make this same design assumption.¹¹

A consistent file name space allows all processes that make up an application to have the same file system view by simply changing directory to the working directory of the invoking application. Consistent user access allows PSE to use the standard UNIX remote shell facility to start up peer processes with standard security checking.

Systems in a common system environment are candidates to become members of a cluster. A cluster is often the largest set of machines running PSE and sharing a common system environment within an organization or site. A cluster is divided into partitions that can overlap. A partition consists of a set of machines grouped together to meet the needs of an application or user. Although partitions may be defined in many ways, systems in a partition usually share common attributes.

Partitions

Parallel programs run most efficiently on a balanced hardware configuration. Typically, organizations have a varied collection of machines. Over time, organizations often acquire new hardware with different network adapters, faster CPUs, and more memory. Such situations can easily lead to increasing difficulty in predicting application performance if scheduling and load-balancing algorithms treat all machines in a cluster equivalently. In addition to hardware differences, individual machines can have different software installed that affects the ability to run applications.

The PSE engineering team recognized that the number of characteristics that users might want to manage for processor allocation and load-balancing purposes would be overwhelming. To limit the problem, a design was chosen that allows machines to be grouped arbitrarily into named partitions. A partition can be thought of as a parallel machine. Although a system can be a member of two different partitions and therefore cause overlap, PSE does not attempt to load balance or schedule processes beyond partition boundaries. Overlapping partitions can therefore create a complex and potentially conflicting scheduling situation. Well-defined and managed partitions allow for flexibility and predictability.

In addition to identifying machine membership, partition definition allows various execution-related

characteristics to be set. Examples include the specification of a default communication type, the default execution priority, the upper bound on the execution priority, and access control to partition resources. Access control is enforced only on PSE-related activity and does not affect the use of the machine for other applications.

Configuration Database

PSE cluster configuration information is captured in a database. The database includes a list of cluster members, partitions, and partition members. Additional attributes such as the default partition of a cluster, user access lists for a partition, and preferred network addresses for members of a partition can be encoded in the database.

The PSE configuration database can be distributed to all cluster members in two ways: by storing it in a file that is accessible from all cluster members, or by storing it as a Domain Name System (DNS) database. The usage patterns of the cluster database fit well with the usage patterns of a DNS database. In particular, DNS provides central administrative control with version numbering to maintain consistency during updates. It is designed for query-often, update-seldom usage; it is distributed and allows secondary servers to increase availability. Applications linked with the PSE run-time libraries transparently access the database to obtain configuration information.

In the DNS database, each PSE configuration token-value pair is stored as DNS TXT records. The original specification for DNS did not have TXT records, but additional general information was attached to domain names at the request of MIT's Project Athena.¹² The list of the TXT records, along with DNS header information such as version number, forms a DNS domain whose name is the PSE cluster name. To facilitate the creation and setup of a PSE cluster, we built the `psdbedit` utility for editing and maintaining configuration databases.

A simple file that is available on all members of the cluster can also be used as the cluster configuration database. The file could be made available through NFS or copied to all nodes using `rdist`. This alternative might be appropriate for very simple clusters where the services of DNS are not warranted or in cases where local policy precludes the use of DNS.

Dynamic Information and Control

In addition to the static information of the configuration database, there are also several pieces of dynamic information that optimize usage of clusters and partitions. At the most fundamental level is availability, i.e., is a machine running? Other information includes the number of CPUs, load average, number of allowed PSE jobs, and number of active PSE jobs. All these

factors can help an application choose the best set of members for parallel execution. This dynamic information is collected by a daemon process (*farmd*). The *farmd* daemon process executes as a privileged (root) process on each cluster member and listens for requests on a well-known cluster-specific UDP/IP port.

Multiple cluster members defined in the configuration database are designated as load servers. The load servers are the central repository for the dynamic information for the entire cluster. Their *farmd* process periodically receives time-stamped updates from the individual daemons. Applications query the load servers for both static and dynamic information. Applications do not themselves parse the database nor query the individual *farmd* daemons running on each cluster member.

Once PSE is installed and configured, *farmd* is started each time the system is booted. The name of the cluster that *farmd* will service and the number of PSE jobs (job slots) that will be allowed to run are set. The *inetd* facility is used to restart *farmd* in response to UDP/IP connection requests, if *farmd* is not running.¹³ Use of the *inetd* facility to start *farmd* improves the availability of machines to run PSE applications by transparently restarting *farmd* in the case of a failure.

As *farmd* daemons are started, they attempt to establish TCP/IP connections with their neighbors as defined by the PSE configuration database.¹⁴ This process is undertaken by all cluster members and quickly results in a configuration ring whose purpose is the detection of node or network failures. We chose a simple ring of TCP/IP connections because the mechanism is passive, i.e., it relies on the loss of TCP/IP connectivity and does not impose any additional load on the system or network under normal conditions. When connectivity to a member is lost, neighboring cluster members report the member being unavailable. This prevents PSE from attempting to schedule new applications on the failed member.

Failures that do not break the configuration ring, but prevent updated load information from being sent to the load server, are detected by checking the time-stamps on previously received load information. As soon as a "time-to-live" period expires for a particular member's load information, the load servers disable further use of the suspect node. System managers are also able to set the number of job slots to zero at any time, thus disabling the host for new PSE-related activities. This has no effect on currently executing applications.

Pseudo-gang Scheduling

The start-up sequence for a PSE application includes the potential modification of execution priority and scheduling policy. These changes are made in accordance with the user command-line options and/or the default characteristics defined by the PSE configuration database. To allow nonroot UID processes to

elevate scheduling priorities and/or alternate scheduling policies, *farmd* modifies the user process's scheduling priority or policy. Processes scheduled at a high real-time priority using a first in, first out (FIFO) queue with preemption policy achieve a pseudo-gang-scheduling effect. (Gang scheduling ensures that all processes associated with a job are scheduled simultaneously.) This effect occurs because of the scheduling preference given high-priority jobs and because PSE polls for messages for a period of time before giving up the CPU.

Using PSE

Parallel applications are developed for PSE using the Digital Fortran 90 compiler. When the Fortran 90 compiler is invoked with the *-wsf N* flag, HPF source codes are compiled and then linked with a PSE library for parallel execution on *N* processors. After defining a partition in which to run, a PSE application can be run simply by typing the name of the application. The following example shows the compilation and execution of a four-process program called *myprog* on a set of cluster members in the partition named *fast*.

```
csh> setenv PSE_PARTITION fast
csh> f90 -wsf 4 myprog.f90 -o myprog
csh> myprog > myprog.out < myprog.dat &
```

Transparently, PSE starts up four processes on members of the partition *fast*; creates communications channels between the processes; supports redirected standard input, output, and error (standard I/O); and controls the execution and termination of the application. Several environment variables and run-time flags are available to control how an application executes. Figure 3 shows how to use PSE.

PSE Application Model

PSE implements an application as a collection of interconnected processes. The initial process created when a user runs an application is called the *controlling process*. It provides application distribution and start-up services and preserves UNIX user-interface semantics (i.e., standard I/O), but does not participate in the HPF parallel computation. The controlling process usually determines which partition members to use for the parallel computation by getting system load information from a load server and then distributing the new processes across the partition. As an alternative, users can direct computation onto specific partition members.

The controlling process starts a process called the *io_manager* on each partition member participating in the parallel execution. Each *io_manager* then starts one or more application peer processes that perform the user-specified computation. The use of an *io_manager* is necessary to create a parent-child

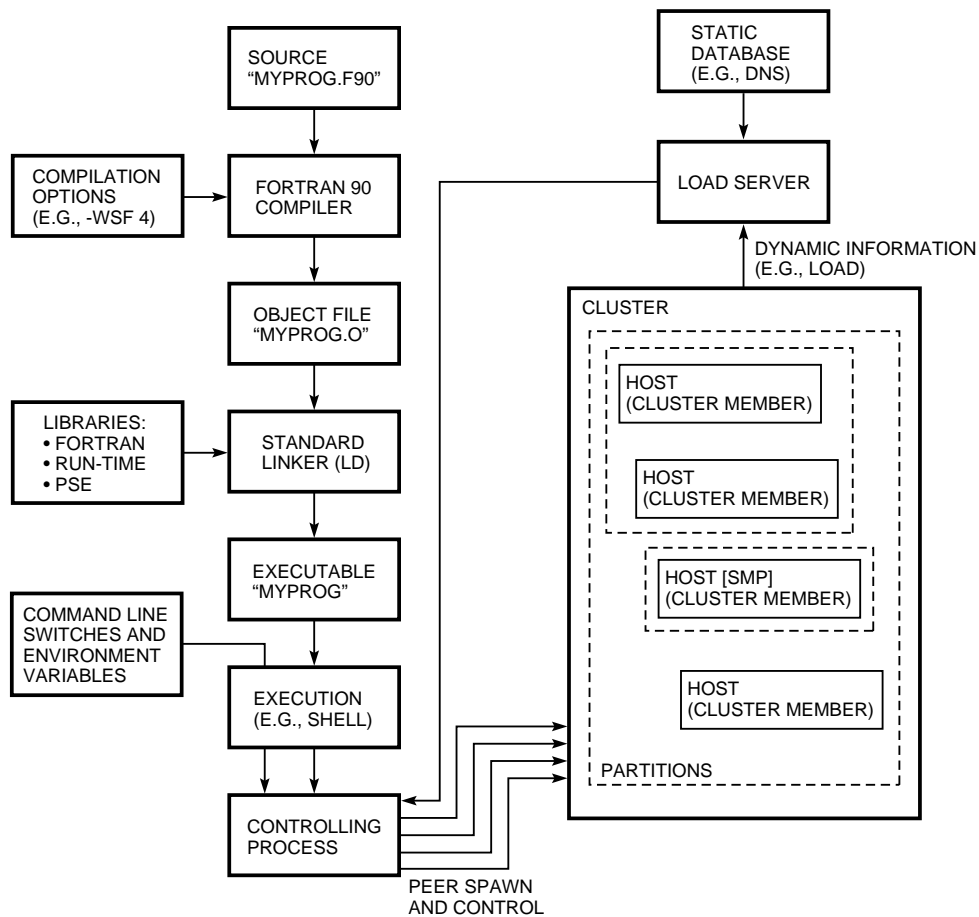


Figure 3
PSE Use

process relationship between the `io_manager` and peer processes. This relationship is used for exit status reporting and process control. It also enables or eases other activities, such as signal handling and propagation. Peer processes create communication channels between themselves and perform standard I/O through a designated peer. Standard I/O is forwarded to and from the controlling process through the `io_manager`. Figure 4 shows a PSE application structure.

Application Initialization

Prior to the execution of any user code, an initialization routine executes automatically through functionality provided by the linker and loader. The initialization routine implements both the controlling process functions and the HPF-specific peer initialization. Because no explicit call is required, parallelHPF procedures can be used within non-HPF main programs, and proper initialization will occur. A simple HPF main program can also be used with PSE to start up and manage a task-parallel application that uses PVM or MPI for message passing.

In general, the controlling process places peer processes onto members of a partition, although hand placement of individual peers onto selected members

is possible. To achieve efficiency and fairness in mapping a set of peers, the controlling process consults with a load server for load-balancing information. Which members are used and the order in which they are used is based on each member's load average, number of CPUs, and number of available job slots.

As an alternative, PSE may map peer processes onto members based upon a user-selected mode of operation. In the default physical mode of operation, PSE maps one peer process per member. In virtual mode, PSE allows more than one peer process per member, thereby enabling large virtual clusters. This is useful for developing and debugging parallel programs on limited resources. Virtual clusters also improve application availability: when the requested number of peer processes is greater than the available set of partition members, applications continue to run; however, they may suffer performance degradation.

Application Peer Execution

Each application peer process has an `io_manager` parent process that provides it with environment initialization, exit value processing, I/O buffering, signal forwarding, and potential scheduling priority and policy modification. Rather than include the

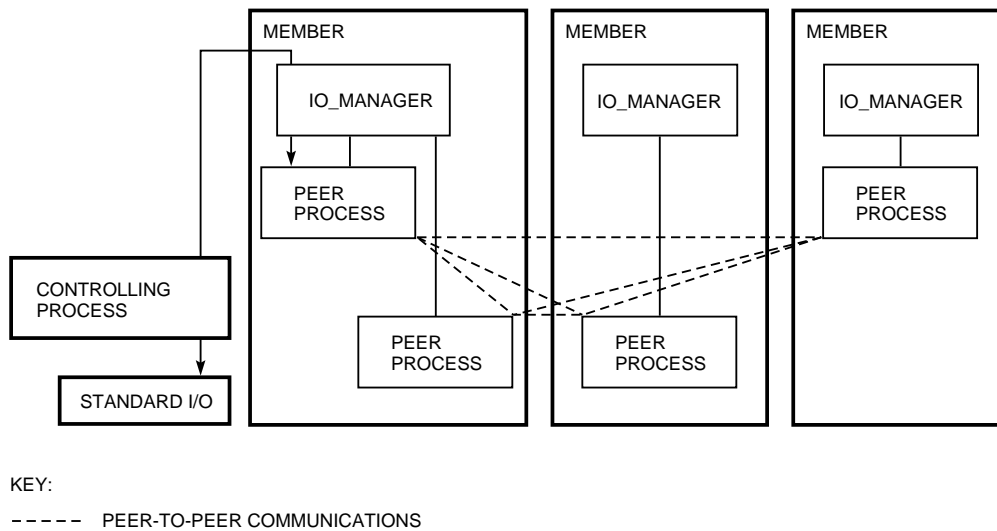


Figure 4
 PSE Application Structure

io_manager's functions in each PSE executable, the io_manager is implemented as a simple utility.

Application peers run the same binary image as the controlling process. They inherit their current working directory, resource usage limits, and an augmented set of environment variables from their controlling process through their parent io_manager. When started, the initialization process described for the controlling process is repeated, but peers do not become controlling processes because they detect that a controlling process already exists. Instead, peer processes return from the initialization routines with communication links established and are ready to run user-application code. Figure 5 represents a controlling process, four application peers running on three members, and the communications between processes.

Application Exit

Multiple peer exits can have potentially conflicting exit values. Coordinating them into a single meaningful application exit value is the most challenging transparency issue faced by PSE. Under normal circumstances, all peer processes exit without error and at approximately the same time. The resulting exit values are reported to the application controlling process by the io_managers. The application (i.e., the controlling process) is allowed to exit without error only when all exit values are recorded and standard I/O connections are drained and closed. The HPF compiler generates synchronization code to guarantee the roughly synchronous exit for all nonerror conditions. This presumption allows PSE to implement a timely exit model, i.e., one by which we can reasonably assume

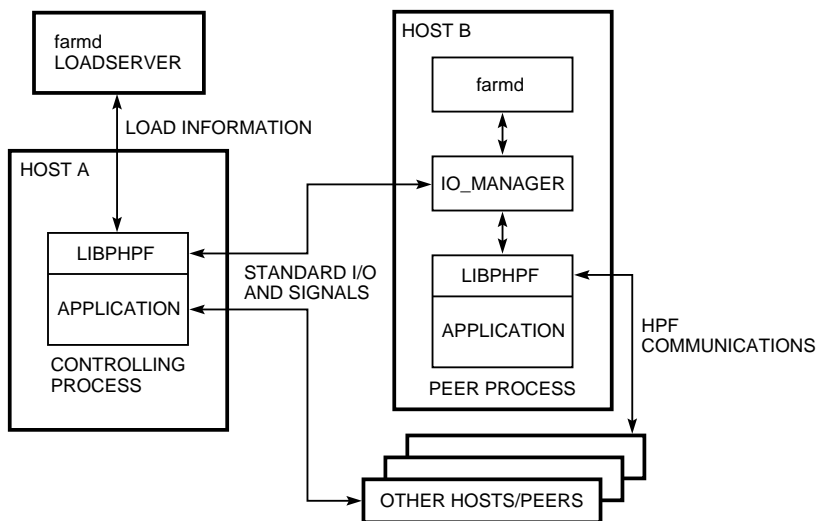


Figure 5
 Communications between PSE Processes

normal activity will cease after receiving the last exit notification from an `io_manager`.

Peers that exit abnormally make it difficult to provide a meaningful exit value for the application. Consider one peer process that exits due to a segmentation fault and another that exits due to a floating-point exception. There is no single exit value possible for the application; PSE chooses the first abnormal value it sees. Furthermore, as a result of error detection in the communication library, the other peer processes will exit with lost network connections. It is possible that the controlling process will see an exit value for this effect before it sees an exit value for one of the causes, resulting in a misleading application exit value. To understand a faulting parallel application running under PSE, the core files associated with each peer process must be examined.

PSE includes support for capturing the entire application core state and for discriminating the multiple core files of a parallel application. Because peer processes share the same working directory, any core files generated would be inconsistent and overwrite one another due to N processes writing to the same core file name. PSE solves this problem by establishing a signal handler that catches core-generating signals, creates a peer-specific subdirectory, changes to the new directory, and resignals the signal to cause the writing of the core file. The root for the core directories can be set through an environment variable.

Issues

Although PSE achieves the standard UNIX look-and-feel for most application situations, complete transparency is not achieved. For example, timing an application-controlling process using the `c-shell`'s built-in time command, does not time user code or provide meaningful statistics other than the elapsed wall clock time to start a parallel application and to tear it down. Another situation that highlights the parallel nature of PSE occurs during application debugging: multiple debug sessions are started by running the application with a debugger flag rather than by using `dbx` directly.

Tools for HPF Programming

The development model for HPF-based applications is a two-step process. First, a serial Fortran 90 program is written, debugged, and optimized. Then it is parallelized with HPF directives and again debugged and optimized. The development tools supplied with PSE address profiling and debugging. Unlike most of PSE, which is language-independent, both the `pprof` profiling facility and the “`dbx` in n windows” debugging facility are specific to HPF programming.

Profiling

Several issues in profiling parallel HPF programs do not apply to Fortran programs that execute serially. HPF execution occurs through multiple processes on multiple processors simultaneously and therefore produces multiple profiling data sets. The storage and analysis of these data sets must be coordinated to produce accurate and comprehensive program profiles. Unlike typical Fortran programs, significant time can be spent communicating in an HPF program. The Digital UNIX `prof` and `pixie` utilities do not handle either of these issues.¹⁵ In addition, the `prof` utility has coarse-grained (1-millisecond resolution) program counter (PC) sampling and reports only down to the procedure level. To address these issues, Digital added profiling support to the Fortran 90 compiler and developed the `pprof` analysis tool.

Data Collecting The PSE parallel profiling facility handles profiling data collection in parallel by writing data to a set of files that are uniquely named. It encodes the application name, the type of data collection, and the peer number of the process. The analysis tool `pprof` merges the data in the file set when performing analysis and producing reports.

It supports two types of data collecting: noninvasive traditional PC sampling and intrusive interval profiling. PC sampling simply records the program counter at each occurrence of the system clock interval interrupt. To achieve an accurate execution profile with PC sampling, programs must be long running to become statistically significant. Also, it is difficult to gather do-loop iteration data using PC sampling.

We developed interval profiling support to overcome the deficiencies of PC sampling. Interval profiling is achieved with compiler-inserted functions that record the entry and exit times for the execution of each event. This produces an accurate execution profile. Events include routines, array assignments, do loops, `FORALL` constructs, message sends, and message receives. Because the entry and exit times are recorded, time spent executing other events within an event is included, which gives a hierarchical profile. To achieve fine-resolution timings (single-digit nanoseconds), the Alpha process cycle counter is used to measure time.¹⁶

Analysis The `pprof` utility provides many different ways to examine and report on a large set of profiling data from a parallel program execution. Different approaches include focusing on routines, statements, or communications. In contrast, `prof` reports on procedures only. With `pprof`, the scope of the analysis can be limited to a single peer process or encompass all application processes. The range of reports generated can be comprehensive or limited to a number of events or

a percentage of time. Users can specify their reports from a combination of analysis, report format, and scoping options. By default, the `pprof` utility reports on routine-level activity averaged across all peer processes, which provides an overall view of application behavior.

Parallel programs execute most efficiently when there is minimum communication between processes. The high-level, data parallel nature of the HPF language reduces the visibility of communication to the programmer. To make tuning easier, `pprof` was designed with the ability to focus tuning on communication. Reports can be generated that help correlate the use of HPF data-distribution directives to observed communication activities.

Debugging

For PSE version 1.0, we are supplying a “`dbx` in `n` windows” capability. Each peer is controlled by a separate instance of `dbx` that has its own Xterm window. This capability gives users basic debugging functionality, including the ability to set breakpoints, get backtraces, and examine variables on an all-peer or a per-peer basis. We added a new command to `dbx`, `hpfget`, that allows the viewing of individual elements of a distributed array. We recognize it as far from meeting the challenges of an HPF debugger, and we are continuing the development of a new debugging technology.

Message-passing Model

One of the goals of PSE is to support high-performance, reliable message passing for parallel applications. At the start of the project, the HPF language and compiler technology were still in their infancy. Even though no HPF application code base existed, the PSE team needed to determine the messaging-passing requirements. To support message passing successfully, PSE had to be flexible enough to accommodate new interconnect technologies and network protocols, adapt to the message-passing characteristics of future HPF applications, and support the changing demands of the compiler. A need for high performance and efficiency with low latency was assumed.

The PSE message-passing facility provides primitives to initialize and terminate message-passing operations, to allocate and deallocate message buffers, and to send and receive messages. A PSE message contains a tag, a source peer number, and variable-length data. The higher layers fill in the tag, which is used as a message identifier on receive. The data is a stream of bytes without any data-type information. These primitives are not intended to be used in the application code. The HPF compiler implicitly generates calls to these primitives. Because the message-passing primitives are tightly coupled to the HPF compiler, overhead such as data-alignment restrictions and error checking can be eliminated.

The PSE message-passing model assumes that the application peers are running on systems with the same CPU architecture and networking capabilities. Each peer process can send or receive binary messages directly to or from any other peer. This is different from the PVM model, where messages might be routed to a `pvmd` daemon to be multiplexed to another peer, or messages might be converted to external data representation (XDR) to allow for data passing between machines with different architectures.¹⁷

Buffer allocation and deallocation routines are specific to each of the communication options that PSE supports. (These options are discussed in the following sections.) Before a message can be sent, a buffer must be allocated. The send primitive sends the message and implicitly deallocates the buffer. The receive primitive implicitly allocates a buffer containing the newly arrived message. Receive buffers have to be deallocated explicitly after they are used. Our initial design allowed a received message buffer to be reused for sending a new message, possibly to a different peer. This design was inefficient, especially when a communication option such as shared memory optimizes buffer allocation on a peer-by-peer basis. The current design uses a peer number as a parameter to the buffer allocation routine and does not allow reuse of the received message buffer.

The send primitive sends a message contained in a preallocated buffer to a specified peer. It guarantees reliable in-order delivery of messages. For underlying protocols, such as UDP/IP that do not provide this level of service, the message-passing library must provide it. A broadcast primitive is also provided to send a single message to all peers.

The receive primitive uses a particular message tag to receive a message with a matching tag from any peer. This allows the compiler to use functions that can perform calculations correctly when data is required from several peers, regardless of the order in which messages arrive. The normal operation for receive is to block the receiving peer until a matching tagged message arrives. A nonblocking receive is also provided to poll for messages.

Communication Options

PSE provides applications with several run-time selectable communication options. Within a single SMP system, PSE supports message passing over shared memory. On multiple system configurations, PSE supports network message passing using the TCP/IP or UDP/IP protocols over any network media that the Digital UNIX operating system supports. Currently, PSE supports a single communication option within an application execution, but the design supports multiple protocols and interconnects. Run-time selection of the communication options and media, which

is implemented using a vector of pointers to functions within a shared library, provides flexibility to introduce new protocols and media without having to recompile or relink existing applications.

Shared-memory Message Passing

The use of shared memory as a message-passing medium allows for very high performance because data does not have to be copied. When designing shared-memory messaging, we looked at a variety of interrelated issues, including coordination mechanisms, memory-sharing strategies, and memory consumption. The use of locks (i.e., semaphores) in the traditional manner to coordinate access to shared-memory segments proved problematic. For example, clients often request a message from any peer, not from a particular peer. This implies the use of a general receive semaphore that senders would unlock after delivering data. Contention for a single lock could be significant and could become a performance bottleneck. Instead of locks, a simple set of producer and consumer indexes is used to manage a ring buffer of messages. Senders read the consumer index and update the producer index, and receivers read the producer index and update the consumer index to synchronize. No locking is required.

Several memory-sharing strategies are possible: all peers may share a single large segment, each pair of peers may share a segment, and each pair of peers may have a pair of unidirectional segments. The use of unidirectional pairs of shared-memory segments offers several advantages: it simplifies the code by eliminating multiplexing; it fits in well with the design of MEMORY CHANNEL hardware, which is unidirectional; and by creating receive segments with read-only protection, it promotes robustness.¹⁸ A disadvantage to the use of unidirectional segment pairs is increased memory use due to limited sharing. Because of its advantages and because the coordination of the producer/consumer index does not require segments to be shared between peers, we selected unidirectional pairs of shared-memory segments as our memory-sharing strategy.

To enhance performance, a receiver spins, waiting for a peer to produce a message. If there is no data after a number of spin iterations, the receiver voluntarily deschedules itself. The number of spin iterations was chosen to be small enough to be polite, but large enough to permit scheduling when a peer produced a message. An additional performance enhancement allows the user, via command line option, to prevent peers from migrating between processors, which results in better cache utilization.

TCP/IP Message Passing

TCP/IP is the default communication option. It provides full wire bandwidth for peer-to-peer communication with large message transfer sizes across a variety

of network media. The implementation of the message-passing primitive operations is relatively straightforward since TCP/IP provides reliable, in-order, connection-oriented delivery of messages. The TCP/IP initialization routine sets up a vector of bound and connected socket descriptors, one for each peer. These sockets are used to send messages to other peers. The receive primitive uses a blocking `select()` system call on all sockets. Because TCP/IP is connection based, abnormal peer termination and network faults can be detected by connection loss.

Although TCP/IP provides acceptable bandwidth, latency-sensitive applications might suffer from the processing overhead of the TCP/IP protocol. The connection-oriented nature of TCP/IP also requires the application to maintain many socket descriptors, which reduces scalability and necessitates the use of expensive `select()` system calls on receive.

UDP/IP Message Passing

To address the latency and overhead of TCP/IP, PSE provides UDP/IP as an option that can be selected at run time. UDP/IP is a connectionless protocol that provides unordered, best-effort delivery of messages. Because UDP/IP is connectionless, the initialization function needs to set up a single locally bound socket description for all peer-to-peer communication. File descriptor use is not a scaling issue when UDP/IP is used for messaging.

Reliable in-order delivery of messages is implemented at the library level. Each peer maintains a set of send and receive ring buffers, one for each peer. The ring buffers have producer and consumer indexes to indicate positions in the ring where messages can be read or written. The buffer-allocation primitive allocates buffers from the send ring whenever possible, or from a pool of overflow buffers when the ring is full. The use of an overflow buffer eliminates the need for upper levels to provide flow control or to block sends. The send and receive primitives manipulate the producer and consumer indexes of the send and receive rings. In-order delivery of messages is guaranteed through the use of a sliding window protocol with sequentially numbered messages. For efficiency, piggy-backed acknowledgments are used.

To improve scheduling synchronization among multiple peers, especially when a high-priority FIFO scheduling policy is used, the UDP/IP option uses a nonblocking socket. On receive, it loops calling the `recvfrom()` system call many times before calling the expensive `select()` system call to wait for a message to arrive. Abnormal peer termination and network faults cannot be detected since the socket layer does not maintain a connection state. The UDP/IP option contains a user-specifiable time-out value by which the peer application will exit when there is no socket activity.

The UDP/IP option provides better bandwidth than the TCP/IP with smaller messages and matches the TCP/IP bandwidth at large message size. The user-level latency reduction, however, was less than expected. The next two sections discuss our investigation into ways to optimize the latency of UDP/IP and the performance of the message-passing options.

Optimizing UDP/IP

Our initial approach to improve latency was to reexamine the standard UDP/IP code path within the Digital UNIX kernel for unnecessary overhead. Our idea was to create a faster path, optimized for a UDP/IP over a local area network (LAN) configuration by reducing numerous conditional checks in the path. Although this work yielded some improvement, it was not enough to justify supporting a deviation from the standard code path. An overhaul of the original code path would have been necessary for this approach to gain significant improvement in latency.

UDP/IP provides a general transport protocol, capable of running across a range of network interfaces. We realize the value in retaining the generality of UDP/IP. For optimal performance, however, we anticipate typical cluster configurations being constructed using a high-performance switched LAN technology such as the GIGAswitch/FDDI system.⁵ In such configurations, the IP family of protocols presents unnecessary protocol-processing overhead. A messaging system using a lower-level protocol, such as native FDDI, would offer better latency, but its implementation requires the use of nonstandard mechanisms to access the data link layer directly, which is less general and portable than a UDP/IP implementation.

Based on the above observations, we designed a new protocol stack in the kernel, called `UDP_prime`, to coexist with the standard UDP/IP stack. `UDP_prime` packets conform to the UDP/IP specification.¹⁹ To reduce the amount of per-packet processing and approach that of a lower-level protocol, `UDP_prime` imposes several restrictions on its use. These restrictions optimize the typical switched LAN cluster configurations. To retain the generality of UDP/IP, `UDP_prime` falls back to the standard UDP/IP stack when these restrictions are not applicable.

Restrictions on `UDP_prime`

The LAN nature of the cluster configuration imposes a restriction on `UDP_prime`. Each cluster member has to be within the same IP subnet, which is directly accessible from any other member. With this restriction, routing decision and internet-to-hardware address resolution can be done once for each peer-to-peer connection rather than on a per-packet basis. Per-packet UDP/IP checksum processing can also be eliminated, because intermediate routing is not

involved and the data link cyclic redundancy check (CRC) is sufficient to guarantee error-free packets.

The next restriction is the maximum length of the message. PSE message passing uses fixed-size buffers. `UDP_prime` restricts the maximum buffer size to be the maximum transmission unit (MTU) of the underlying network interface. This eliminates per-message IP fragmentation and defragmentation overhead. Since the messaging clients have to fragment the messages into fixed-size buffers at the higher layer, there is no need for the IP layer to perform further fragmentation.

One complication in our current implementation occurs when multiple peers are running on a single system while others are on remote systems. The default behavior for peers within a single system is to communicate across the loopback interface. In this situation, there are two MTU values, one for the network interface and one for the loopback interface. Our current implementation of `UDP_prime` does not allow communication over the loopback interface so that a single-size MTU can be used. Further studies need to be done to find an optimal maximum buffer size, taking into account multiple MTU values, page alignment, and so forth.

Based on the above restrictions, `UDP_prime` optimizes the per-packet processing overhead of sending a packet by constructing a UDP, IP, and data link packet header template for each peer at initialization. Except for a few fields, the content of these headers is static with respect to a particular peer. `UDP_prime` defines a new IP option, `IP_UDP_PRIME`, for the `setsockopt()` system call, to allow the messaging system to define the set of peers and their Internet addresses involved in the application execution.²⁰ The IP option processing, done prior to sending any message, makes routing decisions, performs Internet-to-hardware address resolution, and fills in the static portion of the header fields. When sending a packet, `UDP_prime` simply copies the header template to the beginning of the packet, minimizing the per-packet processing overhead and increasing the likelihood of the templates being in the CPU cache. Several header fields, such as the IP identification, header checksum, and packet length fields, are then filled dynamically, and the complete packet is presented to the interface layer.

`UDP_prime` Packet Processing

Since a `UDP_prime` packet is a UDP/IP packet, the standard UDP/IP receive processing can handle the packet and deliver it to the messaging client. To trigger the use of `UDP_prime` optimized receive processing, the sending system uses the type of service (TOS) field within the IP header to specify priority delivery of the packet.²¹ The priority delivery indication does not by itself uniquely differentiate between `UDP_prime` and UDP/IP packets, as any other IP packets can also have the TOS field set to priority. As a result, the

optimized receive processing has to check for the packet's adherence to the UDP_prime restrictions. Nonadherence to the restrictions reroutes the packet to the standard receive processing code.

When a packet arrives at a network interface, the interface posts a hardware interrupt, and the interface interrupt service routine processes the packet. The standard interrupt service routine deletes the data link header, and hands the packet over to the *netisr* kernel thread.²² *Netisr* demultiplexes the packet based on the packet header contents and delivers it to the application's socket receive buffer. *Netisr*, designed to be a general-purpose packet demultiplexer, runs at a low-interrupt priority level. The main reason for a thread-based demultiplexer is extensibility. New protocol stacks can be registered to the thread. Since there is no a priori knowledge of the execution and SMP locking requirements of these stacks, a thread-based low-interrupt priority demultiplexer is needed so that the network interrupt processing time can be held to a minimum. The extensibility feature, however, introduces a context switch overhead.

For UDP_prime, the packet header processing time on the receive path is almost a small constant. We modified the interface service routine to demultiplex the packet by processing the data link, IP, and UDP headers, and deliver the packet to the socket receive buffer without handing it over to *netisr*. This short circuit path is used only when the packet is a UDP/IP packet with no IP fragmentation and with priority delivery indication. If these conditions are not met, the standard *netisr* path is chosen. The UDP_prime receive path eliminates the *netisr* context switch overhead. This is a significant advantage, especially when the receiving application runs with a real-time FIFO scheduling policy.

SMP Synchronization

One difficulty in designing the UDP_prime stack to run in parallel with the standard UDP/IP stack was in SMP synchronization.²³ The socket buffer structure is a critical section guarded by a complex lock. Requesting a complex lock in Digital UNIX blocks execution if the lock is taken. To prevent deadlocks, its use is prohibited at an elevated priority level, such as the case in the interrupt service routine. To work around this problem, a new spin lock was introduced in the short circuit path and in the socket layer where access to the socket buffer needs to be synchronized.

Performance

To measure message-passing performance, we used two DEC 3000 Model 700 workstations connected by a GIGAswitch/FDDI system using TURBOchannel-based DEFTA full-duplex FDDI adapters. Each work-

station contained a 225-megahertz (MHz) Alpha 21064 microprocessor and was running the Digital UNIX version 3.0 operating system.

Figure 6 shows the message-passing bandwidth for TCP/IP, UDP/IP, and UDP_prime transports at different message sizes. The bandwidth was measured at the message-passing application programmer interface (API) level, taking into account allocation and deallocation of each message buffer in addition to the data transmission. TCP/IP, UDP/IP and UDP_prime bandwidth peaks at approximately 95 megabits per second at a 4,224-byte message, approaching the FDDI peak bandwidth. UDP/IP approaches the peak bandwidth at a 1,400-byte message, and UDP_prime at a 1,024-byte message. Reaching the peak bandwidth using small messages is a measure of protocol processing efficiency.

Figure 7 shows the minimum message-passing latency for TCP/IP, UDP/IP, and UDP_prime transports at different message sizes. The latency was measured as half of the minimum time to send a message and receive the same message looped by the receiver system over many iterations. The measurement made allowance for the allocation and deallocation of each message buffer, in addition to the round-trip transmission.

Compared to the TCP/IP option, UDP/IP has a slightly higher minimum latency. This is not expected, because the original goal of the UDP/IP option was to reduce TCP/IP processing overhead. It is, however, encouraging to see only a slight degradation in latency when the reliable in-order delivery protocol is implemented at the library level. This prompted us to use the same protocol engine in the library for UDP_prime. At a very small message size (4 bytes),

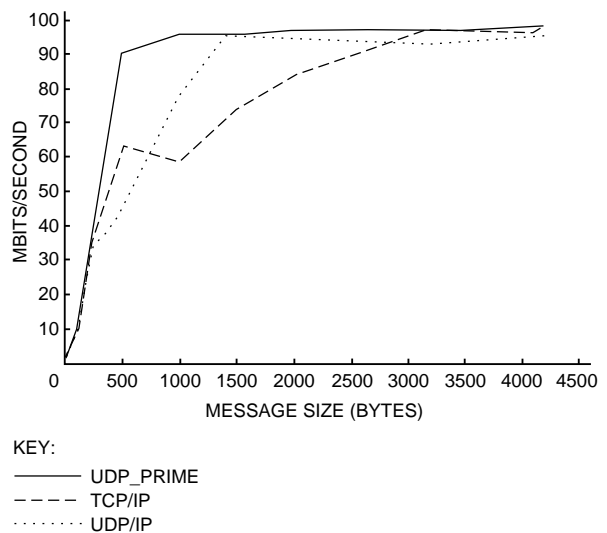


Figure 6
Peer-to-Peer Bandwidth

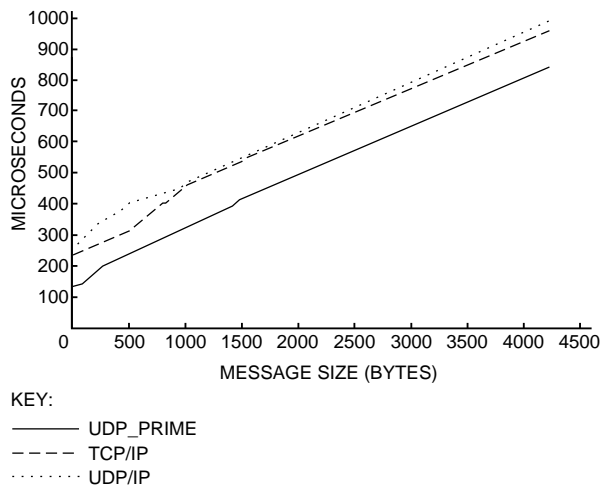


Figure 7
Minimum Peer-to-Peer Latency

protocol processing overhead dominates the latency. At this point, UDP_prime was 44 percent (103.5 microseconds) better than TCP/IP, even though UDP/IP and UDP_prime use the same mechanism.

As the message size increases, the protocol processing time remains constant, but the data copy time becomes dominant. Despite this, UDP_prime was approximately 12 percent better at a 4-kilobyte message.

Future Work

The current communication options along with the UDP_prime optimization provide good performance for HPF-style message passing on SMP systems and clusters. To remain competitive, however, we need to consider support for new high-performance communication media and configurations. We are working on support for MEMORY CHANNEL, the use of multiple interconnects and protocols within an application running on a cluster of SMPs, and lightweight protocols for use with ATM at speeds of 622 megabits per second and higher. The flexibility of the message-passing design will allow current applications to use future communication options without relinking.

We are also working on a new HPF debugger technology. Debugging a cluster-style HPF program is considerably harder than debugging a uniprocessing program. HPF's single-program multiple-data (SPMD) parallel programming model includes a single-threaded control structure, a global name space, and loosely synchronous parallel execution. HPF also supports the calling of extrinsic procedures that use other parallel programming styles or nonparallel computational kernels.

The goal of an HPF debugger is to present the application in source-level terms. Since HPF is roughly Fortran 90 with data-distribution directives, HPF is conceptually a single-threaded application with the compiler transforming pieces of the application to execute in parallel. As a result, an HPF debugger has to take the states from the actual peer processes and recreate a single source-level view of the application. It is not always possible to do this with complete precision. Consider the user interrupting the application, which interrupts the peer processes at different points within the computation. It is unlikely each peer is at the same place (e.g., the same program statement), and it is quite likely that the stack backtraces of the peers differ! Even if they are at the same place, they could be in different iterations of their local portions of a parallelized loop-like operation.

At the start of the HPF debugger project, we surveyed a variety of debuggers and disqualified all of them for logistical and/or technical reasons. Rather than modify an existing debugger technology so that it could debug cluster-style HPF programs, we initiated an effort to build a new debugger technology. As we continue to design the new HPF debugger to be general-purpose, portable, and extensible, we will be able to capitalize on modern programming concepts, paradigms, and techniques.

Summary

PSE contains the tools and execution environment to debug, tune, and deploy parallel applications written in the HPF language. From an end user's perspective, PSE provides transparency, flexibility, and compatibility with familiar tools. Using standard UNIX command syntax, the same executable can be run serially or in parallel on hardware ranging from a single-node system to a cluster of SMP systems. PSE supports several high-performance message-passing protocols running over a variety of network media. From a system administrator's perspective, PSE provides the flexibility to create a cluster from standard components and to control the cluster by assigning access controls and setting scheduling policy and priorities. Although it currently supports only the HPF language, PSE has the flexibility and generic infrastructure to support other parallel languages and programming models.

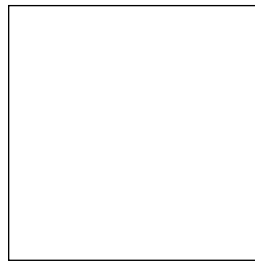
Acknowledgments

The PSE team would like to thank the members of the Fortran 90 and HPF compiler teams and to acknowledge the contributions of Chuck Wan, Rob Rodon, Phil Cameron, Israel Gale, Rishiyur Nikhil, Marco Annaratone, Bert Halstead, and George Surka.

References

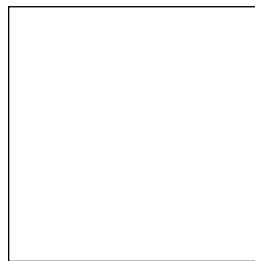
1. *Digital High Performance Fortran 90: HPF and PSE Manual* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-2ATAA-Te, 1995).
2. G. Bell, "Scalable, Parallel Computers: Alternatives, Issues, and Challenges," *International Journal of Parallel Programming*, vol. 22, no. 1 (1994).
3. H. Kung et al., "Network-based Multicomputers: An Emerging Parallel Architecture," *Proceedings Super-Computing 91*.
4. W. Michel, *FDDI: An Introduction to Fiber Distributed Data Interface* (Newton, Mass.: Digital Press, 1992).
5. R. Souza et al., "GIGAswitch System: A High-performance Packet-switching Platform," *Digital Technical Journal*, vol. 6, no. 1 (Winter 1994): 9–22.
6. Internet Engineering Task Force, "Domain Name System," *RFC 883* (November 1983).
7. J. Harris et al., "Compiling High Performance Fortran for Distributed-memory Systems," *Digital Technical Journal*, vol. 7, no. 3 (1995, this issue): 5–23.
8. G. Geist et al., *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing* (Cambridge, Mass.: The MIT Press, 1994).
9. W. Rosenberry, *Understanding DCE* (Sebastopol, Calif.: O'Reilly & Associates, Inc., 1992).
10. W. Gropp et al., *Using MPI: Portable Parallel Programming with the Message Passing Interface* (Cambridge, Mass.: The MIT Press, 1994).
11. *LSF Administrator's Guide* (Toronto, Ont., Canada: Platform Computing Corporation, 1994).
12. G. Champine, *MIT Project Athena: A Model for Distributed Campus Computing* (Newton, Mass.: Digital Press, 1991).
13. W. Stevens, *UNIX Network Programming* (Englewood Cliffs, N.J.: Prentice-Hall, 1990).
14. D. Comer, *Internetworking with TCP/IP* (Englewood Cliffs, N.J.: Prentice-Hall, 1991).
15. *DEC OSF/1 Programmer's Guide* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-PS30C-TE, 1993).
16. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, Mass.: Digital Press, Order No. EY-L520E-DP, 1992).
17. Internet Engineering Task Force, "XDR: External Data Representation," *RFC 1014* (June 1987).
18. R. Gillett, "Memory Channel Network for PCI: An Optimized Cluster Interconnect," *Hot Interconnects* (1995).
19. J. Postel, "User Datagram Protocol," *RFC 768* (Menlo Park, Calif.: SRI Network Information Center, 1980).
20. *DEC OSF/1 Reference Pages, Section 2: System Calls* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-PS30C-TE, 1993).
21. J. Postel, "Internet Protocol," *RFC 791* (Menlo Park, Calif.: SRI Network Information Center, 1981).
22. Open Software Foundation, *Design of the OSF/1 Operating System* (Englewood Cliffs, N.J.: Prentice-Hall, 1993).
23. J. Denham, P. Long, and J. Woodward, "DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation," *Digital Technical Journal*, vol. 6, no. 3 (Summer 1994): 29–43.

Biographies



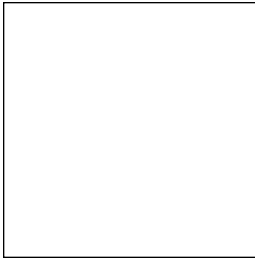
Edward G. Benson

Ed Benson is a principal engineer and the project leader for the parallel software environment product. Ed is a 1981 graduate of Tufts University. He joined Digital in 1984 after working at Harvard University and ADAC Corporation. In previous work at Digital, he led the DECmpp and VAXlab software projects and contributed to the design and development of the POSIX real-time extensions in Digital UNIX and OpenVMS.



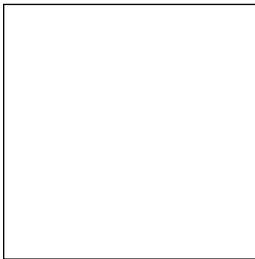
David C. P. LaFrance-Linden

David LaFrance-Linden is a principal software engineer in Digital's High Performance Fortran Group. Since joining Digital in 1991, he has worked on tools for parallel processing and has developed a promising new debugger technology capable of debugging HPF. He has also contributed to the PSE implementation and compile-time performance of the HPF compiler. Before joining Digital, he worked at Symbolics, Inc. on front-end support, networks, operating system, performance, and CPU architecture. He received a B.S. in mathematics from M.I.T. in 1982.



Richard A. Warren

Richard Warren is a principal software engineer in the High Performance Computing Group, where his primary responsibility is the design and development of Digital's parallel software environment. Since joining Digital in 1977, Richard has contributed to PDP-11 systems development and VAX 32-bit shared-memory multiprocessor designs. He has also been a member of Corporate Research, first as an assignee in parallel processing to the Microelectronics and Computer Technology Corporation (MCC), and later as a researcher at the Digital Joint Project office at CERN, where he helped develop high-availability system software. Richard has a B.S. in electrical and computer engineering from the University of Massachusetts and is a co-inventor on several patents relating to coherent write-back cache design and high-performance bus/memory designs for SMPs.



Santa Wiryaman

A senior software engineer in the High Performance Computing Group, Santa Wiryaman develops enhancements to the Digital UNIX kernel and UDP/IP protocol stack to support optimal performance of message passing over FDDI and ATM networks. Since joining Digital's performance group in 1987, he has also contributed to many network-related performance characterizations, benchmarks, and the development of performance tools for UNIX and Windows NT. Santa received B.S. (1985) and M.S. (1987) degrees in computer science from Cornell University and Rensselaer Polytechnic Institute, respectively.