A System Software Architecture for High-End Computing

David S. Greenberg (dsgreen@cs.sandia.gov)* Ron Brightwell Lee Ann Fisk
Arthur Maccabe Rolf Riesen

Abstract

Large MPP systems can neither solve grand-challenge scientific problems nor enable large scale industrial and governmental simulations if they rely on extensions to workstation system software. At Sandia National Laboratories we have developed, with our vendors, a new system architecture for high-end computing. Highest performance is achieved by providing applications with a light-weight interface to a collection of processing nodes. Usability is provided by creating node partitions specialized for user access, networking, and I/O. The entire system is glued together by a data movement interface which we call *portals*. Portals allow data to flow between processing nodes with minimal system overhead while maintaining a suitable degree of protection and reconfigurability.

1 Introduction

The power of the last decade's supercomputers is now available in affordable desktop systems. However, the demand for ever increasing computational power has not abated. For example, the US Department of Energy (hereafter DOE) has recently established a program called the Accelerated Strategic Computing Initiative (ASCI). The goal of ASCI is to enable 100 trillion floating point operations per second (teraflop/s or TF) *sustained* performance on key simulation codes by the year 2005. The DOE's aggressive schedule is due to the need to replace a test-based process for maintaining the nuclear stockpile with a simulation-based process. The change from test-based to simulation-based processes must be completed before the retirement of the aging population of nuclear scientists who can verify the conversion. At the same time industries from oil-and-gas exploration to aeronautics to pharmaceuticals are beginning to discover the enormous power inherent in high-fidelity, three-dimensional simulations.

Although the speed of individual processors has continued to increase by a factor of two every two to three years the capabilities required by the DOE and others clearly necessitates the coupling of large numbers of processors. Sandia National Laboratories (SNL) has recently developed, with Intel, a massively-parallel, distributed-memory machine with a peak performance rate of over 1.8TF (hereafter the Teraflop machine). A sustained computational rate of over 1TF on the standard MPLinpack benchmark was achieved using $\frac{3}{4}$s of the hardware early in 1997.

Recently a variety of applications have been scaled to tackle problems of unprecedented size and complexity. Sandia's CTH code models very high speed impacts. A recent calculation used 100 million grid cells to model the 3-D effects of a missile impacting a target. Shock waves were tracked through a detailed model of missile components in order to determine whether critical fusing components would survive long enough to perform their functions. A second application of CTH was to simulate the impact of a one-kilometer comet into the Atlantic ocean. The simulation ran for 48 hours and showed that large quantities of ocean water would be vaporized by the tremendous energy of the impact and ejected into suborbital ballistic trajectories that reenter worldwide. The result would be devastating tidal waves and a cloud of water and debris enveloping the globe that would affect the Earth's climate.

The success of Sandia's Teraflop machine has been in part due to its system software architecture. Neither the distributed workstation model of full-service, Unix-like OSs communicating via sockets and TCP/IP nor the shared memory model of a single multi-processor OS image meets the needs of such machines. Instead, Sandia developed a new model in which the vast majority of nodes run a simple, light-weight kernel. These compute-node kernels cooperate to scalably (using a logarithmic tree) load an application onto a group of nodes, each kernel assigns physical memory to the application and sets up the appropriate non-paging virtual

*Sandia National Laboratories, Mail Stop 1110, P.O. Box 5800, Albuquerque, NM 87185-5800. This work was supported by the U.S. Department of Energy and was performed at Sandia National Laboratories for the U.S. Department of Energy under contract DE-AC04-94AL85000.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

memory. Once the program is loaded the kernel is only active during message passing and to handle errors. The result is that applications are able to utilize a large fraction of the available CPU and memory resources without competing with the system software for time or space.

In order to supply the non-compute intensive needs of users the model includes additional node partitions. A service partition supplies the standard Unix image to the user. This partition supplies login, shells, status, and tool interfaces as well as the ability to launch parallel applications into the compute partition. I/O partitions supply disk and file services and access to off-machine networks such as ATM and HiPPI. The partitions responsible for file services must be scalable to allow disk and tape services to grow with compute speed and must maintain a parallel file system in order to allow delivered I/O bandwidth to grow as well.

Processing nodes, both within partitions and between partitions, are glued together with a data movement abstraction called *portals*. The purpose of a portal is to provide a flexible mechanism whereby an application, library, server, or, in general, task can describe how its memory can be accessed by remote nodes. In its simplest form a portal maps to a block of the user's virtual memory space. The system software can quickly and safely determine the memory destination of an incoming message and allow DMA hardware to move the data directly from the interconnection network into the proper physical memory.

Requests for system services and status messages between compute and service nodes can be built on portals without the need for polling daemons. Tools such as debuggers and performance monitors can tie "scalability elements" in the compute partition to user interface functionality in the service partition without requiring the application to be actively involved. Parallel file systems and I/O libraries in general can organize data within the compute partition before it is moved to the I/O partition. Pull protocols can be used to prevent the massive buffering and/or the overflows which often occur when I/O streams are sent from many compute nodes to a few I/O nodes. Visualization software can gather only the data needed for the current view rather than having to download all data to a workstation.

The entire intent of the system software architecture is, thus, to allow system software to provide a complete environment for application developers and application users. Different needs such as user shell, compute intensive kernels, I/O, and visualization can be met by tailored pieces of system software. This functional partitioning is supported by a novel view of inter-node data movement which is designed to allow flexible, efficient transfer without the need for buffering or heavy-weight protocols. The resulting system can take advantage of tightly-coupled, reliable networks in ways which are not practical with traditional TCP/IP-like glue.

2 Crews of Nodes

Many system architectures have derived their name from naturally occurring self-organizing phenomena such as hives of bees, swarms of piranha, or tornadoes. The architecture developed at Sandia is better understood in rough analogy to a building construction job. The system software plays the role of a general contractor it gathers together crews of workers (nodes) and provides them with the proper tools (message passing primitives, address protection, etc.) to do a specialized task (run a compute intensive application, supply a parallel file system, or create a scalable user interface). Each work crew is expected to provide coordination appropriate to its task rather than relying on general, system-wide coordination.

Sandia has implemented two versions of system software around these general ideas, SUNMOS [19] on the Intel Paragon and Puma [31, 28] on the Paragon and Tflops systems. Currently, the Puma system is being moved to a commodity-based, distributed supercomputer which is code named Computational Plant. The details of how to best realize the general architecture are still being worked out but several general principles are clear:

1. Individual nodes should be as independent as possible. It should be possible for each node to function without any interaction with other nodes. One consequence of this principle is that system software on one node should communicate with system software on another node only when absolutely required. Examples of required communication are the creation of work crews (loading of applications), the notification of faults, and the provision of services directly requested by the user such as attaching a debugger.

2. Highest application performance requires that system software limit its use of CPU and memory resources to the greatest extent possible. In some cases this may mean making the underlying hardware more visible to the user than is recommended by proponents of transparent, portable systems. In

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

Section 3 we describe early experiences with the Intel Paragon which led to the decision to override the desirable attributes of transparency.

It is not possible to achieve the necessary performance without giving users more control over the hardware. User control allows performance-savvy applications to manage the hardware in aggressive ways which would not be safe for the system software to attempt. Transparency is restored for less aggressive users by adding an additional layer, such as a runtime library, which hides the hardware and provides interfaces and abstractions specific to the given user. The strategy of providing as direct access as possible to the hardware and supplying transparency through user-level libraries enables the desired high-level of performance and has the side effect of making the underlying OS simpler and easier to manage.

3. Applications, servers, and services with distinct system requirements should be supported by distinct system partitions. On a parallel machine the partitions can consist of distinct nodes and thus system software can be simplified within each partition to provide only the needed services. The role of partitions is discussed in more detail in Section 4.

4. Efficient, user-level data movement mechanisms are essential for both intra-application performance and inter-service coordination. Sandia developed *portals* to serve this purpose. In Section 5 the attributes of portals which are essential to the work crew model are described. Details of the implementation of portals can be found in [28].

Clearly not everyone will agree with these principles. In Section 7 comparisons are made to other existing systems. We believe that the ultimate arbiter will be application programmers who will choose systems which are convenient to use and which supply high performance.

3 Early Lessons

The genesis of the Sandia system architecture was collaborations with nCUBE and Intel to make their early massively parallel machines (i.e. the nCUBE2, the iPSC, and the Paragon) meet Sandia's needs. Unfortunately, the use of full Unix-based systems* on each node of these systems created severe resource constraints. In response to the resource limitations SUNMOS [19] was developed. SUNMOS was designed to be a light-weight kernel providing applications with efficient access to CPU and memory resources. The emphasis on resource limitations led to innovation on several fronts: memory models, message buffering, and system services. Although subsequent machines from Intel, Cray, IBM, and others have removed some of the resource constraints there are still lessons to be learned from working within the early constraints.

Local disks and paging Very large scale distributed memory systems often do not have a disk local to each node. Demand paging often created unacceptable bottlenecks. SUNMOS, and subsequent Sandia systems, thus implemented a non-demand paged virtual memory model to provide address space protection without the overhead and non-predictable performance of paged systems. Even on systems such as the IBM SP or workstation clusters which do have local disks, the move away from demand paging has significant advantages. The biggest advantage is that both user-level and system-level data movement routines can avoid overheads due to checking whether a page is or is not in memory. In addition, the cascading of delays due to many nodes waiting for a page fault to be serviced on one node are avoided.

The cost of not having demand paging is in transparency to the user as to the size of physical memory. Users must know an upper bound on the amount of memory to be used. When multiple processes share a node they must divide up the physical memory (in current Sandia systems each process specifies a desired maximum heap size). As systems mature it may be possible to provide some form of paged memory. However, given the cost and rarity of very large systems it is often a better us of the resource to quickly run many problems which fit into physical memory than to use a smaller fraction of the CPU power available in solving one larger problem.

*The nCUBE 2 ran Vertex, an assembly coded small kernel on each compute node.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

Buffering Common, general-purpose communication protocols such as TCP/IP attempt to provide reliable message transfer regardless of physical conditions. Implementations often make use of buffers assigned to each sender/receiver pair. On early MPPs the allocation of a message buffer for each possible sender quickly consumed all available memory. Even on today's machines with 100s of MBs of memory per node the cost of allocating buffers to potential connections is exorbitant. SUNMOS adopted the "comm buffer" approach (as in nCUBE's Vertex [21]) whereby all incoming messages are deposited in a single heap-organized common buffer. An optimization for pre-posted messages was implemented which provided a kernel trap to modify the otherwise read-only comm buffer and thus avoid some unnecessary copying. However, the common buffer approach lacked flexibility and still necessitated frequent copies.

As will be described in Section 5 Sandia has moved the management of buffering up into the user application. Libraries such as MPI hide some of the management effort but again some degree of transparency is lost when users directly manage the buffers. The advantages, at least to library writers, described later in this paper make it worth while to assume the responsibility for buffer management.

System calls Once the use of full Unix on all nodes had been rejected it was necessary to figure out how to support the plethora of system calls which users had come to expect from Unix OSs. In order to save resources (both on machines and of the development team) it was necessary to exclude all but the most critical system functions from SUNMOS. Unsupported functions were split into two groups: disallowed and indirected. Many functions such as system status calls and external interfaces such as sockets were deemed unnecessary and their inclusion led to either compile/linker errors or to exception errors at run time. A few functions such as file access and process control (i.e. kill) were implemented through an interface to a full-function Unix kernel.

The Unix-kernel was a variant of OSF 1/AD [33]. A rudimentary communication mechanism was inserted into the OSF kernel to allow a rolling queue of messages to be maintained between OSF and SUNMOS. Process kill requests could be sent out to all the nodes involved in a program and the local SUNMOS kernels could kill the processes. File requests could be passed from SUNMOS nodes to OSF nodes which in turn controlled disks. Data read from the disks could then in turn be passed back to the SUNMOS nodes.

Unfortunately, the OSF to SUNMOS interface was not particularly robust. Messages could be lost, timing errors were common, and bandwidth intensive operations such as file I/O were choked. It was clear that a cleaner model was necessary and thus the partition model described in the next section was incorporated into Sandia's second generation OS, Puma.

4 Functional Partitions

Functionality in SUNMOS was partitioned in order to meet resource constraints. In Puma, functionality is partitioned in order to match logically similar functionality to tailored system software. The partitioning of system software has the following advantages:

1. A reduction in the conflicts between performance needs and feature needs.
2. The ability to separately scale the resources applied to various functions. Not all functions require the power of all the nodes.
3. The ability to dynamically change the types of services provided and/or the relative amounts of resource applied to each service.

In the remainder of this section we describe three natural partitions: service, I/O (disk, tape, and network), and compute. For each partition we describe the particular functions required, the performance implications, the scaling needs, and some possible implementation strategies. Then after a discussion of important features of Portals in Section 5 we discuss how the various partitions can be tied together in a robust and efficient manner.

Service Partition The requirements of the service partition are the closest to those of a standard workstation. Users must be able to login, run programs, provide input, receive output, debug, tune performance, and in general interact with the computer. It has been our experience that some common functions such as text editing and compiling are best performed off the high-performance machine since they do not require the

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

high-performance compute power and are best done as close to the user as possible.[†] The functions of the service partition are, in general, those functions of a workstation which when moved into the high-performance realm involve the interaction of serial, user-directed interfaces with parallel application codes.

In our architecture these service functions are decomposed into an *interface component* which runs in the service partition and a *scalability component* which runs in the compute partition. Most interface components currently run as sequential programs on a single service node. We expect, however, that some functions such as debugging will need to implement the interface component as a small-scale parallel or distributed process. The scalability component typically is replicated on each of thousands of compute nodes.

For system-wide services such as program load the scalability component is part of the compute-node operating software. These components are gathered in a user-level portion of the operating system which we call the *process control thread* or PCT. The system is designed to allow new PCTs to be loaded on subsets of the nodes without rebooting the system. Thus new scalability components can be tested on a running system and if necessary the PCTs can be tailored to particular users' needs.

Scalability components which are tied to a particular application are implemented as libraries which are linked with an application before it is loaded into the compute partition. This approach allows developers to tailor the system to their needs without having system-wide privileges.

I/O and Network Partition The I/O partition may contain several subpartitions devoted to disk, tape, and networks. These subpartitions will differ primarily in whether they include drivers for storage devices or for network interfaces such as ATM. In all cases, as with the service functions, the functionality is split between a scalability component on the compute partition and a driver/management component in the I/O or network partition.

For disk I/O the scalability component will have to provide a means of moving data from or to compute node memory to or from I/O node memory. On the Tflops system this task is accomplished by libraries linked to the executable. Implementation improvements are continuing with the intention that control information is separated from large data flow and that data flow is coordinated to eliminate large buffers and buffer overflow. Within the I/O partition (and in current implementations in part within the service partition) specialized processes called fyods manage the communication. In addition a parallel file system (which Intel calls PFS) stripes data across multiple disks and provides an interface for user management of parallel files.

It is an area of active research at Sandia to clean up the manner in which I/O partitions are used. We currently make great use of existing facilities such as UFS to manage local disks and Intel's PFS to stripe data across I/O nodes. The system architecture, however, provides a natural way for higher-level, more sophisticated I/O systems such as MPI-IO [20] and Galley [22] to be mapped directly to disk resources. The portal interface can be used to simplify data movement and reordering. We intend to soon develop an I/O system which matches more closely the way in which applications users address their data rather than being tied to rigid mappings between compute node memory and disk arrays. We are also working to develop clean interfaces between parallel entities such as a simulation application and either an I/O partition parallel file system or a parallel visualization engine.

Compute Partition While the functionality provided by service, I/O, and network partitions is necessary for real work to be completed we have devoted the most effort to the compute partition. If highest performance is not achieved on thousands of nodes then it becomes uneconomical to build a large-scale system. As has been mentioned, the compute partition necessarily contains many elements which interface to other partitions. The challenge is to ensure that these elements have as little effect on compute speed as possible.

One way in which the effect of scalability components is minimized is by the fact they are user-level entities managed by user-directed libraries. There is no need, for example, to maintain a daemon polling for potential requests despite the fact that no debugging is going on. Even during debugging the requests can be automatically serviced via the portals mechanism to be described in Section 5. Similarly, if the user never asks for program status then the compute partition will never waste resource sending out status. (The reliability of the system will certainly require some health monitoring. We expect, as in the Tflop machine, that a separate diagnostic system including dedicated hardware will do most of the monitoring.)

[†]Aggressive compilation techniques such as those employed by Tera Computers might require moving compilation to the high-performance machine but in this case compilation will look more like a parallel program than a service function.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

System software's effect on applications is further minimized by allowing portals to overlay user memory. It is not necessary to set aside buffers or to regularly copy data into buffers since the service routines can directly access the memory. At Sandia, where users tend to use every byte available to them (even in the 600+GB Tflop system) this parsimonious memory use can make the difference between a usable service and an unusable toy.

Of course, the system cannot merely not cause problems. It must provide for the computational needs of the application. In Puma, we maintain a very simple memory model. There is no demand paging and applications are allowed to partition the available physical memory between multiple processes. The partition is currently static (decided at load time.) Our user community does not currently require dynamic memory partition but we are studying how to achieve it without affecting performance. The compute-partition system kernel schedules processes as requested by the user-level PCT. It ensures that proper address protection between processes is maintained. Threading within a process may be provided by a user-linked library such as Pthreads [11]. (We have implemented a version of Pthreads but have not yet extensively used it.)

The system also supplies standard libraries such as libmath and libc. Some functions such as I/O are translated into requests to scalability components, some such as malloc are implemented directly, and a few are deemed not interesting in the parallel context and are replaced with stub routines informing the user that the requested function is not available. Determining the correct mixture of responses has been an iterative process between the system designers and the applications users. One of the reasons for our emphasis on flexible library and PCT design is that it is clear that we will never satisfy everyone with a single solution.

While Sandia has had great success using Puma as the compute node operating system we are also interested in evaluating other operating systems for this role. The major question is can full service operating systems such as Linux or NT be sufficiently controlled to allow them to provide similar levels of performance and predictability.

5 Portals

While the partitioning of large-scale parallel machines into crews of nodes as described in Section 2 provides tremendous flexibility and tailorability it does not supply any structure for making the nodes within a crew or multiple crews work together. The structure is provided by a primitive which we call a *portal*.

In its simplest form a portal is, as suggested by the name, a window into the memory of a node. Other nodes can be given permission to write and read data to and from the window. The system software is required simply to read a short fixed sized header, use the destination process name and portal id to determine the physical address represented by the header, validate the address, and stream the data following the header into the memory. For a system, such as the Paragon and Tflops systems in which there is DMA hardware available to stream the data the system overhead can be very small. The advent of smart network adapters such as Myrinet provides the opportunity for the header to be analyzed off processor and the data to arrive without any direct effect on computation. (Of course the use of the memory banks and buses and possible cache effects will still be seen but for current systems this is unavoidable.)

Interaction with a demand paging system can, however, be costly. In the Sandia developed compute partition software we do not allow demand paging and thus can begin streaming data as soon as virtual to physical address translation has concluded. In our implementation of portals within fuller functionality OSes such as Linux we pin down the pages corresponding to memory associated with portals. Other techniques such as copying into fresh pages and swapping for virtual pages later are possible but we have not been willing to pay the cost of maintaining free-lists and correcting for partially filled pages. We are optimizing for the case where the network is very fast and where time not spent pulling data out of the network can quickly cause backups in the network, congestion, deadlock and/or packet loss.

The simple data movement mechanism described above is designed to take maximum advantage of fast network hardware. It also has an advantage over explicitly shared memory. If data comes into a node which is destined for a now dead process or an invalid address it is discarded.

Portals have been designed for MPP networks with the assumption that the hardware detects data corruption and no third party can get physical access to the network. While these assumptions have allowed many optimizations with our system software further research and experience on a wide range of architectures will be necessary to determine how best to trade requirements on network capability against complexity in system software.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

5.1 Advanced features

The simple portal described above provides efficient, safe data movement between nodes but the creation of higher level interfaces between nodes such as message passing with the MPI standard or service-compute partition realizations of debuggers is made simpler by extending the portal definition.

The first extension is to preface a portal's memory window with an optional *matching list.* The idea of a matching list is that the data header includes a few bytes of information which identifies the sender's purpose. For an MPI message the information might include the communicator name (a message context which can be used to isolate library routines), the message tag (an application supplied id used to distinguish between messages which might arrive in arbitrary order), and the sender's node id. The receiver sets up (usually through a library) a list of pairs of match patterns and addresses. The match pattern can include wild-carded bits if desired. The system software traverses the match list looking for an acceptable match. Once a match is found the data is streamed into the associated memory window as in the simple case.

While the match lists can be arbitrarily long they do not have as serious an impact as say demand paging. The size of the matchlist is controlled entirely by the user. Thus if the length is excessive the user can fix the problem. We are working on real-time system software which will include MPI libraries which prevent the user from creating long lists and kernel adaptations which limit the time spent traversing such lists. If necessary we will consider allowing more complex matching data structures such as trees or hash tables.

The second extension is to allow the memory window to have internal structure. A simple block window allows the sender or the receiver to specify an offset from the beginning of the window. A contiguous block of data is then transferred to or from this location. An independent block allows the window to be organized as a list of buffers. Successive messages are placed into successive buffers. Thus, a double buffering scheme can easily be achieved. For example, a logical array can be updated for the next iteration by neighboring nodes while the home node is using the array to compute the current iteration. Since the user controls the portal structure the buffer pointers can be swapped before moving to a new iteration.

A combined block also organizes the window as a list of buffers but messages are placed in buffers as if the buffers were concatenated to form a single large window. This organization allows a scatter operation to be simply expressed. The final organization we have implemented is a dynamic or heap window. As messages arrive the memory window is treated as a heap and buffers are allocated from within the window and inserted into a list. All memory for the list pointers and heap buffers is drawn from the receiver defined window and the receiver can thus traverse the list at a later time. The dynamic portal is especially useful for handling unexpected messages within message passing protocols.

A slight variant on the window organization is the ability to save the incoming message header, header and data, or data only. Saving the header alone and having the system discard the data allows speculative protocols to be implemented. Data can be sent with the hope that the receiver has prepared room for it. If the hope is correct, as determined by matchlist use, then the data is placed in the waiting buffer. If the timing is wrong then the receiver need only save the small header and can request the data to be resent at a later time. Saving the data alone and having the system discard the header allows data to flow directly into working arrays. No space need be left for headers and successive messages can be concatenated.

A third extension to portals is to allow handlers to be associated with portals. Many message passing APIs allow a handler to specify an action in response to a message. Typical actions might be to increment a counter or to reply with an acknowledgment. The special case in which the action is to send back the data in some portion of the portal window allows get or pull operations. At Sandia we are currently examining how to provide improved flexibility to handlers. We wish to balance the efficiency of having handlers executed in the system software kernel with the security of having handlers execute in the user's address space.

5.2 Portals as organizers

Portals are more than just a mechanism for moving data. They represent a way for applications to manage the flow of data. Rather than just supplying buffers to smooth out temporal or speed mismatches the portal organizes data. Many systems supply handlers [25, 30] which can be associated with certain message events such as message arrival. A completely general handler system will, of course, allow the data to be organized. However, issues of deadlock and interleaving of handler instructions with network interface instructions can be quite complex and performance may suffer.

Portals, on the other hand, offer the application the ability to use data structures to organize data rather than using active procedures. Data structures are used to communicate to the kernel what needs to be done

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

with incoming messages. Setting up these data structures is very cheap, since no trap into the kernel is necessary. On message arrival, the kernel (or a routine on a co-processor) examines the data structures to decide what actions to take. Even if the user process is not currently active or is busy with another task, it is possible to handle the incoming message quickly – the data structures are in place to control the fate of incoming messages.

Some of the data type constructors within MPI try to achieve similar results but suffer from being implementation dependent add-ons rather than fundamental building blocks. Specific examples of how portals can organize data in tying together partitions are given in the next section. Here we try only to give the general context.

A common situation is that one node maintains information within some high-level language data structure. The information might be arrays of physical values at grid points, current pruning bounds, a work queue, performance data read from hardware counters, or scheduled disk blocks to be written. Another node needs some of the information and has sufficient understanding of the data structures to know where in the data structure its data will be found. Using handlers the retrieving node might send a message which tells the handler what is wanted and where to send the data. The handler then interprets the request, packages the data, and sends it out. With portals the requester can make use of the data structure itself and simply request the data directly without any interpretation. In a homogeneous system with DMA support the data may be retrieved without interrupting the processor holding the data.

6 Building on Portals

To demonstrate how portals can be used to organize communication and facilitate the coordination within and between crews we describe several library routines which we have designed and prototyped.

6.1 Communication within the compute partition

MPI has become the data movement library of choice for applications writers within DOE and throughout the world. [7] MPI's success has been in part due to the fact that it standardized industry practice in a portable syntax. High-performance users, however, still worry that performance will not be portable. Based on our implementation of MPI for the Intel Tflop machine [2] we believe that portal-like low-level data movement can indeed provide high performance for MPI.

We were able to concentrate our efforts on performance due to the two-level, portable implementation of MPI from Mississippi State University and Argonne National Laboratory, MPICH [9]. MPICH divides its implementation into a top level which translates the multitudinous MPI API into calls to a small number of low-level routines called the Abstract Device Interface (ADI). The ADI, where most of the performance issues reside, must then be implemented on each target system.

Many important operations for the MPICH ADI mapped directly to features of the original portal design. In a few cases we made use of the open-ended portal design to add in new features which made MPI more efficient. Some of the uses of portal features are detailed below.

1. MPI provides two mechanisms to label messages in order to provide a safe communication space and allow for message selection within that space. The first mechanism is the *communicator* which provides a context and a grouping of nodes. The system is responsible for insuring that messages stay within the context and the group of nodes. The second mechanism is a user-defined *tag*. The system propogates the tag to the receiver where it can be ignored or used for message selection by the application.

   The portal match list offers a direct means to provide both contexts and tags. The kernel uses the match bits in the message header and the receiver's portal structure to determine where to put an arriving message. The MPI library can encode the potentially quite complex expectations defined by many users issuing many prospective receives (IRECVs) into the portals and match lists. The kernel merely follows a few pointers and is led to the correct location for the incoming data.

2. Since many current high-performance systems are memory-bus bandwidth limitted it is crucial that efficient implementations of MPI avoid copying data within local memory. The MPI syntax provides a nonblocking receive call which allows the application to tell the system where a message should be placed. If the message has already arrived then it is copied to the location. If, however, the application succeeds in issuing the receive before the message arrives then the system can record the location and return control to the user.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

The MPI standard does not require the system to make good use of the provided information. A valid implementation could ignore the provided location until a blocking call was issued to wait for reception. At this point the system could search out the message and copy it to the location requested.

The independent block portal provides a simple way to make use of the nonblocking receive's information and avoid a copy. An IRECV for a message which has not yet arrived is translated into the attachment of an independent block portal to a matchlist with the appropriate context and tag information. The block is superimposed over the location specified in the IRECV. When the message arrives it is deposited directly in the user's desired location without any buffering (assuming the message is received by a contiguous datatype).

3. A correct MPI implementation must be able to handle messages which arrive "unexpectedly", that is, prior to a matching receive being issued. As mentioned earlier an implementation which reserves space for every possible incoming message is not practical in large-scale systems. Instead messages must be kept in some sort of shared buffer. The heap organized dynamic portal is perfect for this purpose. An entry can be placed at the end of a matchlist which directs incoming messages to the dynamic portal heap. The MPI library can the search this heap, *from user-level*, when a later receive call is issued. The kernel processing of "unexpected" messages is quite similar to the processing for pre-posted messages – a match list is traversed, a buffer is allocated in the heap, and the data is directed to the location.

4. Despite attempts to place data directly into user data structures or into common buffer pools it is possible for many senders to flood a single receiver. One way to prevent flooding is to use some sort of handshake mechanism. Unfortunately handshake protocol can add latency to all messages. It would be nice to handshake only when necessary. We have experimented with speculative protocols whereby a message is sent without handshaking on the understanding that the receiver can discard the message if there is no room for it. On distributed systems with low bandwidth links this approach would be ill-advised but on high-performance MPPs it turns out to be a winning approach.

   We make use of the ability of portals to store just headers to refine the speculative protocol. The data portion of a "long" message which arrives prior to a pre-posted buffer can be discarded while only the header is stored. When the destination location becomes known to the library it can use a pull protocol to retrieve the data. The sending process need not wait or poll for requests to send the data. It sets up a *read* portal overlaying the data before the speculative send. When this memory is needed again then it can check for an acknowledgment from the receiver.

5. As just mentioned it is sometimes necessary to send acknowledgements or non-acknowledgements in response to messages. An additional feature of portals is the ability to request the kernel to send an acknowledgement back to the sender of the data. If the sender has set up a portal to receive the acknowledgement then a dangerous cascade of kernel created messages can be avoided.

We believe that the quality and performance of message passing within the compute partition is the most crucial attribute of MPPs and thus have placed the most attention on it. However, portals also facilitate robust inter-partition communication.

6.2 Service to compute control

In small-scale parallel systems it is possible to control the compute nodes by copying instructions to local disks or by serially signalling each compute node. For large-scale systems such as the Tflop machine a more scalable, preferably logarithmic approach is necessary. We illustrate with the task of loading a program onto thousands of nodes and commencing execution.

The program launcher written for the Tflop system takes a standard command line request from the user (along with information about how many nodes to use and other MPP specific information) and starts the requested program in the compute partition. The launcher must distribute the executable, arguments, and environmental variables onto a set of nodes and interact with the PCTs of those nodes to allocate local node resources and start the processes. To make this load scalable and efficient, a fanout tree is used to distribute code and data segments as well as control messages.

The use of portals facilitates the use of a pull protocol over the fanout tree. As each segment (e.g. text) is available on a node in the fanout tree, the PCT on that node sets up a portal over the portion of memory

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

that contains the segment. The children in the tree can then read from that portal without further action from the parent PCT. While the first segment is progressing down the tree a second segment (e.g. data) can be retrieved by the parent.

This simple, pull-oriented tree greatly simplifies the load routine. Each node needs only to repeatedly get its next segment, setup the outgoing portal, and then process the segment. There is no need to interleave the maintenance of the tree with segment processing.

The program launcher is, of course, part of the base system software. For additions to the system such as PVM virtual machines or debuggers it is important to provide a similarly scalable and robust mechanism to user routines. We have developed a standard library to connect interface and scalability components which we call the *host interface library (HIL)*. The HIL is built on top of portals and provides and abstraction for establishing connections between service partition routines and compute partition libraries. Once the connection is made the HIL aids in passing data and in cleaning up when the connection is closed.

For example, we have partnered with Oak Ridge National Labs to implement a PVM daemon using HIL. PVM daemons are responsible for setting up intermachine links (in our case an ATM connection from New Mexico to Tennessee), launching parallel jobs at both ends of the connection, and helping route messages between to two machines. The HIL maps all of the service to compute communication into portals. We are currently working on ways to allow the portal communication to bypass the daemon and flow directly to ATM nodes.

We have also implemented a parallel debugger using the HIL. User commands and source translation are processed on the service side. Requests for program state and/or control commands pass are translated by the HIL into portal requests to the compute side. Libraries on the compute side have established portals over debuggable memory thereby providing access without copying into buffers. We are currently investigating the possibility of adding a modest scale layer which uses portals to gather, filter, and compress the debug output so as not to swamp the service partition. Eventually we hope to have a general purpose tools library using HIL features to allow debuggers, performance analysis tools, and statistic gatherers to connect to running applications and interact with them in a scalable manner.

6.3 Compute to I/O

Perhaps the weakest area of system software for high-performance computing is the support for I/O to disks and WAN/LAN networks. The portal and partition model provides an opportunity to redress some of the imbalance. We describe here two possible approaches, one for maintaining a parallel file system and one for attaching to ATM networks. Early versions of these designs currently run on the Tflops machine but are hamstrung by the incomplete implementation of portals within the I/O partition OS.

Network gateways Sending data under the portal design across an external network link is simple since the receiver is depended on to organize the data. The send can thus be transparently extended to flow through network gateway nodes. If data needs to go out over a network connection it can be routed (either by a PVM library or some other communications package) to a "pseudo-portal" at a node in a network partition. The network node can recognize the portal identifier in the header as special, pass the header up to the network routing routine and hold the body for forwarding on the network. The routing routine can prepare the network connection by, for example, placing header information onto the wires. It can then request that the body of the message by released to the network card.

A message can pass through many of these network forwarding nodes until it reaches one which recognizes the portal as local to its machine at which point the body is routed to the destination node where it is processed as if it was generated locally. This approach is similar to the PacketWay approach.

If the data is destined for a device which expects TCP/IP or some other high-level protocol rather than to a portal then additional packaging is necessary. The portal/partition approach will allow us to experiment with various places to do the packaging. On the one hand, the sender can process a protocol stack locally and send a fully packaged message to the network node. The network node may add routing information and then forward as in the portal case. On the other hand, the sender could send just the data to the network node which would receive the data will a normal portal, process a protocol stack and then send the message out over the external network. The first case is nicely scalable but requires lots of library code on the compute node. The second case allows the protocol stacks to be maintained in one place but may create performance bottlenecks. The correct mix will have to await experimentation.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

Parallel disk usage We imagine systems with thousands of compute nodes and at least hundreds of disks. The flow of data between compute nodes and disks can be unpredictable and bursty. The most efficient order of the data within compute node memory can be different from the best order of the data on disks. The disk system can include RAID redundancy and hot spares. Disk may be added to the system over time. Unix-style file systems are clearly not well suited to handling such large volumes of data and complex data uses. Several groups implemented parallel file systems to address this problem (see MPI-IO, PFS, SIO, Galley). While these libraries address the organization of the data and sometimes schedule the movement of data they typically rely on some lower level mechanism (UFS perhaps) to do the data movement.

On the Paragon and Tflops machines, Intel and Sandia has developed an I/O model which attempts to provide third-party control of I/O via a parallel file system daemon along with unmediate data flow between compute nodes and disk nodes. As mentioned above, there are technical reasons why portals cannot be fully utilized on these systems. We describe here an I/O model which makes full use of portals.

- It is assumed that users specify the number of compute nodes desired and the amount of I/O resource desired (either by specifying a number of I/O nodes or by specifying a parallel file system with implied striping over certain I/O nodes.)
- Each compute node is assigned a controlling I/O node and portal at program load time. The assignment may be done statically or the compute-node I/O library may modifying the assignment over time.
- I/O control requests such as opens and closes are written to the controlling portal and handed a return portal name. The controlling I/O nodes either maintain distributed status or pass the control requests to central resources. In either case, responses are sent directly back to the return portal. High performance compute-node libraries will include collective calls which reduce the number of control requests made.
- Data movement such as reads and writes will be performed by setting up a portal over the appropriate memory in the compute node and sending a request to the controlling I/O node. The request can be translated into individual actions for I/O nodes with attached disks. Each disk node can perform its data movement independently by using the return portal and appropriately calculated offsets.
- The I/O-node libraries can rearrange data, use prefetch strategies, and in general try new techniques to improve disk performance independently of the compute nodes.
- Compute-node libraries can reblock data whenever the application provides sufficient hints to do so.
- Applications have control over the compute node memory and can choose to copy data to I/O buffers for asynchronous processing or to organize computation so that algorithmic data structures can serve as copyless buffers.

6.4 Organized data is easily moved

The recurring theme of the preceding examples is that efficient, flexible communication and coordination can be accomplished by overlaying a portal on naturally organized data. Flow control can be provided by having small groups of nodes pull data from larger groups of nodes. Robustness occurs automatically since the control of the data is encoded in the local memory as opposed to in control messages.

7 Related Work

It is an exciting time to be designing system software as standard techniques such as demand paging and process-based contexts are being challenged by micro-kernel, nano-kernel, and even embedded firmware approaches. In this section we do not attempt to survey all the new avenues in operating systems but instead touch on a few key concepts which have developed over the last 10 to 15 years: host machines growing into service partitions, light-weight kernels providing a route to efficiency and flexibility, demand paging as a way to manage memory, and techniques to avoid the cost of copying data in the presence of deep memory hierarchies.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

Functional partitions Vertex on the nCUBE 2 [21] is a minimal operating system that sparked the idea for SUNMOS. Vertex and the nodes of the nCUBE 2 are controlled by a Unix workstation that serves as a host. The host is responsible for allocating nodes and loading executables. It also serves as a gateway between the users and their programs running on the compute nodes. I/O nodes with disks can be attached to the nCUBE 2. There is no partitioning in the sense described in this paper. Nodes are assigned a specific role at hardware installation time. The communication interface between the host and the compute nodes is different from the one used to communicate between compute and I/O nodes.

Many other systems exist that use a host computer to control a set of compute nodes. Examples are Thinking Machines' CM-5 [24], Intel's iPSC [13], and a multitude of research systems, such as Carnegie Mellon's C.mmp [32]. In all these systems, the compute partition can grow to meet increasing demands, but a single host remains responsible for the control and access of the whole machine.

The Intel Paragon was one of the first machines that was *host-less* and allowed the system administrator to tailor the size of the service partition to the number of users and processes present on the system [14]. Intel's OSF 1/AD operating system provides a single system image that gives users the illusion of a single computer, no matter how many service nodes are configured. The drawback of this particular implementation is that the failure of a single node requires the reboot of the whole machine to ensure that the OSF 1/AD kernels on all nodes have an updated system status and are in sync with each other.

Minimal kernels Minimal operating systems can be easier to tune for performance, validate for reliability, and customize for multiple uses. Several groups have demonstrated significant performance improvements by implementing operating systems which are designed to be as small as possible [17, 18]. Other groups have built minimal kernels in order to demonstrate that it is possible to improve reliability and maintainability by proving (or at least arguing) that a (very) small core has been implemented correctly (see for example [26] and [10]). A third set of groups has concentrated on increasing the flexibility of their systems. It is often the case that no single abstraction of the hardware is proper for all users. The necessity of providing a single, general-purpose, one-size-fits-all abstraction for all users is replaced with the ability to customize the kernel to each user's need. An extensible kernel allows a user process to customize the kernel thereby creating streamlined, efficient, and process-specific interactions between user-level and kernel-level functions [8].

In some situations, such as embedded systems, the appeal of minimal OSs is strengthened by resource constraints and the need for predictability. One example of a system which takes advantage of these properties is QNX [10].

For high performance computing we rely on small kernels to provide efficiency, reliability, low resource overhead, and flexibility. In addition we have found that these properties can be extended to produce a scalable system.

Virtual memory A classic example of resource management is the concept of virtual memory. Early computer system and programming paradigms were constrained by the need to conserve a few tens of KB of memory. Virtual memory decoupled the application memory from the physical memory and demand paging used disks to simulate large "core" memories. While the effects of demand paging have been studied extensively for workstation class machines [16, 27], fewer studies have measured the effects of running the same program in parallel on thousands of nodes.

At first glance, a system, such as the IBM SP-2, with a disk attached directly to each node would seem to decompose the parallel paging problem into independent paging tasks which are well-understood. However, local disks may not be desirable from a cost, reliability, and managability stand point. Systems such as the Intel Paragon gather the disks onto a small number of IO nodes. The distribution of pages from these nodes to many compute nodes can be quite difficult. Intel has created paging trees that dampen the impact of all nodes of an application requesting their first page to execute a program. While clever algorithms help, users still experience intolerable slow-downs when thousands of nodes run out of physical memory and start swapping to disk.

Furthermore, even having a local disk does not solve the problem of parallel paging. Efficient use of thousands of processors requires careful coordination. If one processor has scheduled the CPU for an unrelated task then many processors can be left ideal. Similarly, one paging processor can cause a vastly magnified delay. Ideally, the processes should be CPU gang scheduled as well as memory gang scheduled [4]. In addition to impacting performance the use of demand paging and a full-featured Unix on each node can

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

cause a loss of predictability [3]. This is especially troublesome for the realtime community but also has a performance impact for other applications where gang scheduling and quick, predictable responses are necessary.

Copy avoidance When the bandwidths of networks started to match and exceed the bandwidth of memory buses in high-end systems, it became clear that even a single memory-to-memory copy is detrimental to achieving the available hardware bandwidth [5]. Mainstream network technology is following in the footsteps of high-performance, massively parallel systems and has uncovered the same problem in clusters of workstations. Software solutions [30, 23, 12] as well as hardware solutions [1, 6] are being investigated to get the full hardware performance while still providing a usable API.

8 Future Directions

We have described an approach to allow system software to be scaled to thousands of nodes while maintaining the flexibility to provide both highest compute efficiency and convenient functionality. In conjunction with Intel this approach has been applied to Sandia's ASCI red Tflops system. The Tflop machine contains over 9000 Pentium-pro processors organized as two processors per node and two nodes, two network interface chips, and one mesh router chip per board. The boards are connected in a $32 \times 34^{\ddagger} \times 2$ mesh using 400MB/s links. A compute partition containing over 4000 nodes runs Intel's Cougar[§] OS which includes a full implementation of portals and the MPI library. A small service partition and a small I/O partition run a variant of the OSF 1/AD Unix OS. A partial version of portals has been included in the OSF kernel in order to allow cross partition communication. Production level computations have been being performed since May 1997.

The use of Cougar in a production system is an important milestone for our approach to scalable system design but a lot remains to be done. We are currently working on a version of Linux which will include a full implementation of portals. The Linux version will allow us to continue work in which we dynamically create partitions with desired OS functionality/performance trade-offs. We have a project with Steve Chapin at UVA to examine how applications can migrate from one OS to another as their functionality needs evolve [15].

We are actively examining both disk I/O and external networking. We are using the power of portals to allow parallel I/O operations involving thousands of compute nodes to be properly marshaled to tens of I/O nodes. The marshaling not only can minimize flow control overhead it can be used to reorganize the data into formats which are more appropriate to users' visualization needs than to the compute partition's node layout. We are working with Oak Ridge National Laboratories to combine the distributed computing power of PVM with Puma and portals. We hope that PVM daemons running in a service partition can manage portals in the compute partition so that data can be sent directly to nodes containing ATM interfaces. With some help from ATM board vendors we hope to have the data flow directly into TCP/IP accelerators and out onto OC12 speed connections.

9 Acknowledgments

We would like to thank the people who have worked on Puma over time and contributed ideas: Jeffrey Bowles, Steve Chapin, William Davidson, Nicolas Droux, Michael Levenhagen, Chu Jong, Lance Shuler, T. Mack Stallcup, and David van Dresser. We thank Phil Papadopoulos for his work on porting PVM and the PVM daemon. We also acknowledge the help of the SC'97 program committee and the reviewers. We thank them for their input.

References

[1] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21th International Symposium on Computer Architecture (ISCA)*, pages 142 153, Apr. 1994.

[2] R. Brightwell and L. Shuler. Design and implementation of MPI on Puma portals. In *Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.

[‡]This dimension is configurable

[§]Recall that, as evidenced by the synonymous names, Cougar is Intel's version of Puma.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

[3] C. P. Brown, R. A. Games, and J. J. Vaccaro. Real-time parallel software design case study: Implementation of the RASSP SAR benchmark on the Intel Paragon. Technical Report MTR 95B0000095, MITRE, July 1995.

[4] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. Technical Report TR 1244, COMPUTER SCIENCES DEPARTMENT, UNIVERSITY OF WISCONSIN, MADISON, WI, Sept. 1994.

[5] C. M. Burns, R. H. Kuhn, and E. J. Werme. Low copy message passing on the Alliant CAMPUS/800. In *Supercomputing '92: Proceedings*. pages 760–769, 1992.

[6] J. B. Carter, A. Davis, R. Kuramkote, C.-C. Kuo, L. B. Stoller, and M. Swanson. Avalanche: A Communication and Memory Architecture for Scalable Parallel Computing. In *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[7] L. Clark, I. Glendinning, and R. Hempel. The MPI message passing interface standard. Technical report, Edinburgh Parallel Computing Centre, The University of Edinburgh, 1994.

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In SOSP 15 [29], pages 251–266.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[10] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126. USENIX Association, April 1992.

[11] POSIX system application program interface: Threads extension [C language] POSIX 1003.4a draft 8. Available from the IEEE Standards Department.

[12] J. S. B. III. A fast mach network IPC implementation. In *Proceedings of the Usenix Mach Symposium*, November 1991.

[13] Intel Corporation. iPSC system overview, [1] 1986.

[14] Intel Corporation. *Paragon OSF/1 User's Guide*, September 1992.

[15] Katramatos et al. Cross-operating system process migration on a massively parallel processor. Submitted to Supercomputing '97, 1997.

[16] C.-H. Lee, M. C. Chen, and R.-C. Chang. HiPEC: High performance external virtual memory caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation(OSDI)*, pages 153–164, Monterey, CA, Nov. 1994. Published as ACM Operating Systems Review, Winter 1994.

[17] J. Liedtke. On μ-kernel construction. In SOSP 15 [29], pages 237–250.

[18] J. Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, Sept. 1996.

[19] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994.

[20] MPI-IO: a parallel file I/O interface for MPI. The MPI-IO Committee, April 1996. Version 0.5. See WWW http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps.

[21] nCUBE system software release 3.0 manual. Distributed by nCUBE Corporation with their nCUBE2 hardware, 1991.

[22] N. Nieuwejaar and D. Kotz. The Galley parallel file system. *Parallel Computing*, 23(4), 1997.

[23] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Supercomputing '95: Proceedings*, 1995.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE

[24] J. Palmer and G. L. S. Jr. Connection machine model CM-5 overview. *IEEE*, pages 474–483, 1992.

[25] P. Pierce. The NX message passing interface. *Parallel Computing*, 1993.

[26] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, London, July 1990.

[27] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Balck, W. J. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computer*, 37(8):896 908, Aug. 1988.

[28] L. Shuler, R. Riesen, C. Jong, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceedings of the Intel Supercomputer User's Group Conference*, June 1995.

[29] *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, Dec. 1995. Published as ACM Operating Systems Review, SIGOPS, volume 29 number 5.

[30] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, pages 256 266, Gold Coast, Australia, May 1992. ACM Press, New York. Published as ACM Computer Architecture News, SIGARCH, volume 20, number 2.

[31] S. R. Wheat, A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup. PUMA: An operating system for massively parallel systems. *Scientific Programming*, 3:275–288, 1994.

[32] W. A. Wulf, R. Levin, and P. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, 1981.

[33] Zajcew et al. An OSF/1 UNIX for massively parallel multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*, pages 449–468, Jan. 1993.

Proceedings of the ACM/IEEE SC97 Conference (SC'97)
0-89791-985-8/97 $ 17.00 © 1997 IEEE