PUMA: An Operating System for Massively Parallel Systems

STEPHEN R. WHEAT[1], ARTHUR B. MACCABE[2], ROLF RIESEN[1], DAVID W. VAN DRESSER[3], AND T. MACK STALLCUP[4]

[1] *Massively Parallel Computing Research Laboratory, Sandia National Laboratory, Albuquerque, NM 87185-5800*
[2] *Department of Computer Science, The University of New Mexico, Albuquerque, NM 87131-1386*
[3] *Department of Computer Science, The University of New Mexico, Albuquerque, NM 87131-1386*
[4] *Intel Supercomputer Systems Division; on-site at Sandia National Laboratories*

ABSTRACT

This article presents an overview of PUMA (Performance-oriented, User-managed Messaging Architecture), a message-passing kernel for massively parallel systems. Message passing in PUMA is based on portals—an opening in the address space of an application process. Once an application process has established a portal, other processes can write values into the portal using a simple send operation. Because messages are written directly into the address space of the receiving process, there is no need to buffer messages in the PUMA kernel and later copy them into the applications address space. PUMA consists of two components: the quintessential kernel (Q-Kernel) and the process control thread (PCT). Although the PCT provides management decisions, the Q-Kernel controls access and implements the policies specified by the PCT.
© 1994 by John Wiley & Sons, Inc.

1 INTRODUCTION

Application programmers developing programs for massively parallel (MP) machines must manage three types of resources: processor cycles, memory, and communication. In addition, they may need access to other types of services (e.g., file system and network connections). The relative demands for these resources may vary from application to application. Some applications (e.g., factoring) may require large numbers of processor cycles, but have relatively low memory and communication requirements. Other applications have significant requirements for all three types of resources.

This article describes PUMA (Performance-oriented, User-managed Messaging Architecture), an operating system for MP distributed memory systems. PUMA is a joint project between the Parallel Computing Sciences Department at Sandia National Laboratories and the Computer Science Department at the University of New Mexico. The PUMA project was initiated in January 1991 with the goal of developing an operating system that would be compatible with Vertex (the vendor supplied operating system for the nCUBE-2) and could be used to explore alternate message-passing schemes. In August 1991, we completed an initial implementation of this operating system.

Received April 1994
Revised May 1994

This work was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract DE-AC04-76DP00789.

Scientific Programming, Vol. 3, pp. 275–288 (1994)
CCC 1058-9244/94/040275-14

In January 1992, the team undertook the design and implementation of a new operating system, PUMA. Early in the design process we identified six goals for the PUMA effort:

1. PUMA would be developed for MP environments, i.e., environments with thousands of processor nodes in a tightly coupled, reliable communication network.
2. PUMA would be portable across MP distributed memory machines.
3. PUMA would be developed to support scalable, performance-oriented applications, i.e., applications that could be scaled to consume all of the resources provided by an MP environment.
4. PUMA would provide a reliable and robust environment for the development of applications.
5. PUMA would provide an open architecture for the development of application-level libraries, i.e., it must be possible to develop efficient, user-level library routines to implement any message-passing paradigm.
6. The initial development of PUMA would emphasize efficiency over functionality.

Like many of the operating systems developed for distributed processing environments (e.g., Amoeba [1], Chorus [2], Mach [3, 4], and V [5, 6]), the PUMA architecture is based on a message-passing kernel. However, unlike many of these systems, PUMA has been developed for an environment in which the communication network is trusted and controlled by the kernel. Whenever a kernel receives a message, it knows that the message was accepted and transmitted by a kernel running on another node in the system. This avoids the need to authenticate messages and simplifies many of the tasks that need to be performed by the kernel and application processes.

In the next section, we describe the programming environment and model that provide that basis for the PUMA design. The third section introduces the basic architecture of PUMA, consisting of the quintessential kernel (Q-Kernel) and the process control thread (PCT). The fourth section describes portals, the communication structure. The fifth section describes the communication control polices and mechanisms embedded in the PCT and Q-Kernel. The sixth section describes the structure of the Q-Kernel and the seventh section describes the structure of the PCT. In the eighth section we compare the structures of PUMA to other systems with similar goals. In the final section we summarize the results of running PUMA on a nCUBE2 and an Intel Paragon.

2 BACKGROUND—THE PROGRAMMING ENVIRONMENT AND MODEL

The PUMA programming environment is based on the host node model. In this model, application programmers do not interact directly with the nodes of the MP processor. Instead, they interact with a host node. A host node program is provided to launch application programs and provide interactive I/O services. Figure 1 illustrates the host node model.

We based PUMA on the host node model so we could concentrate on the features needed for the efficient execution of MP applications. In particular, we were able to ignore many of the issues related to general purpose, multiuser systems (e.g., user authentication, process scheduling biased to meet the needs of interactive users, the need to support graphical user interfaces, etc.).

Although PUMA is currently based on the host node model, we recognize the benefits to be gained by providing application programmers with a unified programming environment. We believe that the best way to achieve this unification is to start with an environment developed for the efficient execution of MP applications, extending this environment as needed to support the features needed for interactive use and application development. In this respect, the approach taken in the development of PUMA is in contrast to the approach taken in the development of OSF/1 AD for the Intel Paragon.

In PUMA, an application consists of multiple processes that run on a collection of processor nodes. The activities performed by the processes can be described by a single program (the SPMD—single program, multiple data—model) or they may be described by several programs (i.e., the heterogeneous process model). PUMA supports multiprocessing on the processor nodes, so it is possible to have multiple processes executing on a single processor node.

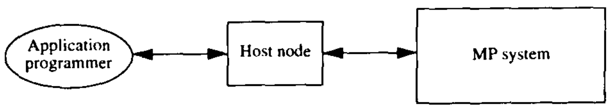

FIGURE 1 The host node model.

When an application programmer launces a PUMA application, the host node program starts by identifying the collection of processor nodes to be used by the application. The host node program then loads the program(s) that describe process behaviors onto the appropriate processor nodes. The application launch is completed when the PUMA nodes have initiated the execution of the processes.

In PUMA, the term *process group* is used to refer to the collection of processes created during the launch of an application program. PUMA provides direct support for explicit message passing among processes. It includes support for intragroup and intergroup communication, as well as communication with server processes.

We distinguish among intragroup communication, intergroup communication, and communication with server processes to reflect differences in expected communication patterns. Intragroup communication is provided as a default. When an application is launched, every process in the application can send messages to every other process in the application.

Intergroup communication is provided to support composibility of applications, i.e., the ability to take the results of one application and feed them directly into the inputs of another application. Intergroup communication rights are not granted as a default, nor are they necessarily granted to all of the processes of an application. To communicate with the processes in a different process group, an application process must first establish a connection to the other process group. Once the connection is established, the application process can send messages to any process in the other process group. Notice that the permitted communication is still fairly restricted: Only the process that initiates the connection can send messages to processes in the other process group. Other processes in the same group as the process that initiated the connection do not automatically gain the right to send messages to processes in the second group. Moreover, processes in the second group do not automatically gain the right to send messages to the process that initiated the connection.

Server processes provide access to shared and/or persistent resources. Communications with server processes are even more restrictive than intergroup communication. The restrictions are intended to provide a higher degree of security for the resources provided by servers. In this case, the application process that establishes the connection only gains the right to send (request) messages to a specific server process, and cannot send messages to all of the processes in a server process group.

3 THE STRUCTURE OF PUMA

The internal PUMA architecture is based on three levels: the kernel, the PCT, and the application/server processes. Figure 2 illustrates the logical structure of a PUMA node.

The Q-Kernel is the lowest level in the PUMA architecture. This level provides basic computation facilities, communication facilities, and address space protection. The PCT is the next level in the PUMA architecture. It provides process management (e.g., process creation and scheduling), naming services (for finding server processes and initiating intergroup communication), and communication capabilities. The server/application process level is the third level in the PUMA architecture. All application and server processes execute in this level.

Each processor node has a Q-Kernel, a PCT, and a collection of server and application processes. Many of the processor nodes may only have application processes. In most configurations, server processes are only present on processor nodes that provide special resources (e.g., disk drives or networking facilities).

The Q-Kernel is responsible for controlling access to the physical resources provided by a processor node. The PCT is responsible for managing access to these resources. For example, the PCT determines the size of the quanta (along with all other scheduling decisions); however, the Q-Kernel enforces execution quanta on all server and application processes. The separation between the Q-Kernel and the PCT reflects a separation between policy and mechanism. The PCT establishes the protection policy and the Q-Kernel

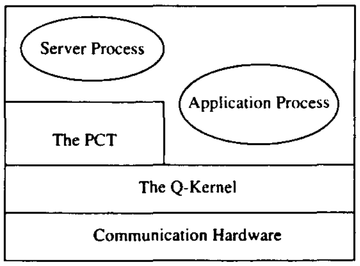

FIGURE 2 The logical structure of a PUMA node.

provides the mechanisms to enforce this policy. Beyond the separation between policy and mechanism, this organization reflects four important aspects of the PUMA design goals: reliability, portability, efficiency, and openness.

3.1 Reliability—Levels of Trust

In developing PUMA, we started with the premise that an MP system is in one of two states: operating or nonoperating. When the machine is the operational state, all of the resources (processors nodes, communication hardware, storage facilities, etc.) are fully functional and available. If a hardware failure makes some of the resources unavailable, the original hardware can be partitioned into one or more independent MP systems with reduced capabilities. However, this partitioning is static and outside of the current scope of the PUMA design.

Given this perspective on hardware reliability, reliability and robustness of the PUMA system are based on our ability to contain the ill effects of erroneous or malicious code. In PUMA containment is based on levels of trust. Each level in the PUMA architecture represents a level of trust. The Q-Kernels running on the different processor nodes trust the communication hardware to provide correct and secure communication between processor nodes. In addition, each Q-Kernel trusts the Q-kernels running on other processor nodes to correctly implement their specified behavior. However, the Q-Kernels do not trust the PCTs, server processes, or application processes. Each PCT trusts the hardware, the Q-Kernels, and the other PCTs; however, the PCTs do not trust the server or application processes. Server and application processes trust the hardware, the Q-Kernels, and the PCTs but do not, in general, trust other application processes.

The trust relation is a partial order. The communication hardware represents the most trusted level—all of the other levels trust the communication hardware. The Q-Kernel is the most trusted level of software whereas the PCT is the next most trusted level. Trust does not represent a total order because application processes do not necessarily trust one another and, as a consequence, are incomparable with respect to trust.

The PUMA architecture ensures that the data structures maintained by one level can only be corrupted by a malfunction in the level itself or a more trusted level. For example, the data structures maintained by the Q-kernel cannot be corrupted by a malfunctioning PCT, server, or application process. To ensure this degree of security, implementations of the PUMA architecture need to provide distinct privilege/protection domains for each level of trust.

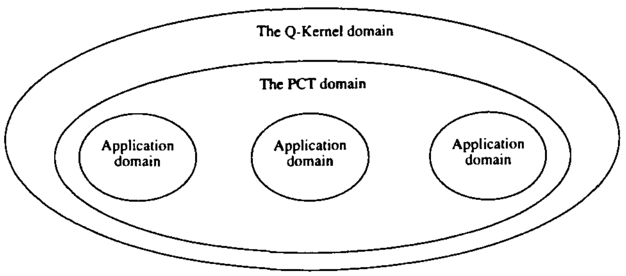

FIGURE 3 Protection/privilege domains in PUMA.

Figure 3 illustrates the relations between the privilege/protection domains in PUMA. The Q-Kernel domain includes all of the physical resources for a processor node. When it begins its execution, the Q-Kernel identifies the memory that it needs for its data structures and maps the remaining memory into the address space for the PCT. Whenever the PCT loads a server or application process, it allocates a portion of its address space for the application process and constructs a new protection/privilege domain.

3.2 Portability

Because the Q-Kernel interacts directly with the hardware, the separation between the Q-Kernel and the PCT reflects different concerns regarding portability of code. Although we expect that a good deal of the Q-Kernel code will need to be modified when we port PUMA to different architectures, we expect that a significantly smaller amount of the PCT code will need to be modified. The PCT is not totally portable. A small amount of the PCT code reflects the address mapping performed by the underlying hardware and this part of the code will need to be rewritten when PUMA is ported to a new MP system.

3.3 Efficiency

In developing PUMA, we noted that control decisions occur far more frequently than management decisions during the execution of an application process. As an example, consider the use of the communication network. Management policies determine the set of processes that a process can communicate with. Although it is unlikely that this set will change frequently, it is likely that a process will frequently communicate with processes in this set. Given this difference in frequency of decisions, the separation between the PCT and the Q-Kernel reflects different levels of concern regarding the efficiency of implementation. In this case, the separation allowed us to concentrate on the efficiency of control activities without needing to consider the impact on management activities.

3.4 Openness

When considering the structure from a more global perspective, we noted that management policies are changed more frequently than control mechanisms. (Alternately, consider the fact that a single set of control mechanisms can be used to implement a variety of management policies.) In this case, the separation between the PCT and Q-Kernel reflects different levels of concern regarding the openness of the system. In the future, we expect to be able to run several different PCTs on the Q-Kernel. For example, one PCT might only provide single tasking whereas another might provide prioritized multitasking.

4 PORTALS

PUMA provides direct support for interprocess communication using explicit message passing. In designing the interprocess communication facilities of PUMA, we sought to minimize the need to use memory copies during communication. As internode communication rates approach (and even exceed) memory copy rates, the need to minimize memory copies has become a critical factor in the efficient use of the resources provided by an MP machine. As an example, we have been able to achieve internode communication rate in excess of 160 Mbytes per second on an Intel Paragon. However, we have only been able to attain memory copy rates of 70 Mbytes per second on the same machine. When communications require a memory copy, the effective throughput drops significantly.

To avoid memory copies and simplify the design, PUMA does not provide any system-level buffering for messages. That is, neither the Q-Kernel nor the PCT provide buffers for holding messages destined for application or server processes. Message passing in PUMA is based on the concept of a portal—an opening in the address space of a process. Most portals are used to store messages sent by other processes. Because incoming messages are written directly into the address space of the receiving process, any need to copy the message body is determined by the application and is not a result of the underlying message-passing primitives.

In addition to portals that buffer incoming messages, PUMA also provides "read memory" portals. When a message is sent to a read memory portal, the Q-Kernel sends a response message to the process that sent the original message. The body of the response message consists of a portion of the memory associated with the read memory portal. Read memory portals can be used to make data available to a collection of processes on an "as needed" basis.

All message transmissions are asynchronous with respect to the execution of application processes. To send a message, the sending process registers a message buffer with the Q-Kernel, specifying a destination process and portal. After the message buffer has been registered, the contents of the message buffer are transmitted to the specified portal of the destination process. The Q-Kernels on the respective processor nodes notify sending and receiving processes when the transmission is complete.

It is possible that a message will arrive when the memory available in a portal is not sufficient to hold the contents of the incoming message. When this happens, the Q-Kernel discards the message and notifies the receiving process that a message was dropped. In particular, PUMA does not provide guaranteed delivery of messages. It is the application's responsibility to provide flow control in portal usage.

Because PUMA does not provide flow control, only applications that need external flow control will be burdened with the additional costs associated with flow control (e.g., the implementation of an RTS/CTS protocol). This approach simplifies the structure of the Q-Kernel and reduces the overhead required for self-synchronizing applications.

PUMA provides four types of portals: kernel managed portals, receiver managed portals, sender managed portals, and read memory portals. The different types of portals are distinguished by the management policy associated with the portal memory.

4.1 Kernel Managed Portals

From the application programmer's perspective, the simplest type of portal is a kernel managed portal. The memory associated with a kernel managed portal is managed as a dynamic heap that is shared between the Q-Kernel and the process.

To use a kernel managed portal, the process needs to initialize and register a block of its memory as a kernel managed portal. After the memory has been initialized and registered, the Q-Kernel will dynamically allocate buffer space in the portal heap when a new message arrives. After the message has been transmitted into the allocated buffer, the Q-Kernel constructs a message header and appends it to a list of received messages. The message header includes important information related to the message body, e.g., identification of the sending process and length of the message body.

Because the message list is maintained in the application's address space, the application is free to process the message list in any fashion that is appropriate within. In most cases, an application process will start by searching the message list, looking for an appropriate message. Because the application performs the searching, application programmers can establish the selection criteria that are most appropriate. When a matching message is found, the application can process the information in the message body without needing to copy this information. When it is done processing a message, the application can remove the message from the message list and release the associated memory, allowing the kernel to reuse this memory for another message.

Figure 4 illustrates the structure of a kernel managed portal. As shown, the message list maintained inside the portal memory starts with list header cell. This header cell is created when the portal memory is initialized. Each element of the list has a header followed by the body of the message.

Kernel managed portals provide the user with flexibility in managing the memory used for communication. The application programmer only needs to predict the maximum amount of space needed for incoming messages. Moreover, because the memory associated with the portal is part of the application's address space, the application can use this memory for other activities when it is not being used to hold incoming messages.

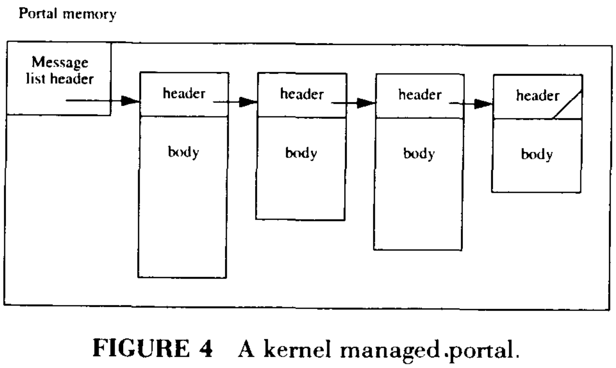

FIGURE 4 A kernel managed.portal.

Although kernel managed portals provide the application programmer with flexibility, they have two drawbacks. First, the fact that the kernel has to perform a dynamic allocation on every message receipt increases the time required for communication, in particular by increasing latency. Second, and perhaps more important, applications may still need to copy message bodies from the portal memory into the data structures used by the application. (This will happen when the application uses contiguous representations, e.g., arrays, instead of linked structures that rely on pointers to determine the actual location of the data.) Because the kernel manages the space used for incoming messages (using dynamic allocation in the portal heap), the application cannot control the placement of the arriving message and may need to copy messages from the portal memory into other application data structures. These difficulties are addressed by the other types of portals provided by PUMA.

Before we turn our attention to the other types of portals, there are two implementation difficulties introduced by kernel managed portals that need to be discussed. First, because the Q-Kernel shares a dynamic data structure, it is possible that the Q-kernel alters the portal structure while the application is manipulating the structure, leaving the application with an inconsistent view of the data structure. Second, because the application process can alter any location in the kernel managed portal, it can leave the message list and/or data structures used to manage the dynamic heap in an inconsistent state. In particular, the application might create a cycle in the message list or the heap free block list, causing the Q-Kernel to enter an infinite loop as it searches one of these lists.

On the Paragon, we handle the first of these problems using the LOCK instruction provided by the i860 processor. When a LOCK instruction is issued, all external events can be disabled for up to 30 instruction cycles. An explicit UNLOCK instruction must be issued within 30 instruction cycles to re-enable external events; otherwise, an internal exception is generated. Because the LOCK and UNLOCK instructions do not require special privilege, we can write user-level libraries that bracket critical codes sections with LOCK and UNLOCK instructions.

Clearly, this solution will not work on machines that do not provide an equivalent of the LOCK and UNLOCK instructions. On these machines, we will implement a "complete/restart" semantics in the Q-Kernel. The idea is to have the application process record its progress through an update activity. The application can be in one three states with respect to an update: noncritical, searching, or updating. Before the Q-Kernel updates a shared data structure, it first checks to see if the application is updating the data structure. If it is, the Q-Kernel completes the update for the application before it begins to access the data structure. After the Q-Kernel has completed its update, it again examines the state of the application. If the application was in the searching or updating state, the Q-Kernel adjusts the restart address for the application. If the application was searching, the Q-Kernel sets the restart address so that the application will restart its search. If the application was updating. the Q-Kernel sets the restart address to the point where the application would have completed its update. This approach is fairly complicated, but is far less complicated and more efficient than providing a general purpose locking capability.

To handle the second problem, we considered the data structures that the Q-Kernel used. There are two data structures that the kernel relies on when it places a message in a kernel managed portal: the message list and the free block list. In the case of the message list, the list header maintains a pointer to the last element of this list. The Q-Kernel uses this pointer when it adds a new message to the end of the message list and, hence, it cannot enter into an infinite loop searching for the end of the message list. Unfortunately, the Q-Kernel must actually search the free block list to find a free block that is large enough to accommodate the incoming message. To avoid any circularity problems in searching this list, the free block list is always ordered by increasing block address. Hence, if the Q-Kernel ever detects a smaller block address while it is searching the free block list, it aborts the search.

4.2 Receiver Managed Portals

The memory associated with a receiver managed portal is managed by the process that owns the portal. To use this type of portal, the receiving process preallocates buffers for the messages it expects to receive from other processes. When messages are sent to a receiver managed portal, they are mapped directly into one of the preallocated buffers.

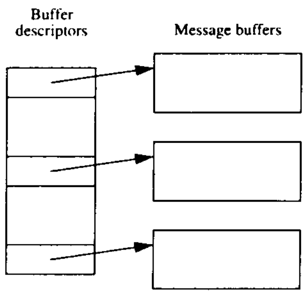

FIGURE 5 A receiver managed portal.

When an application uses a receiver managed portal, it first allocates space for a collection of message buffers and an array of message buffer descriptors—one descriptor for each message buffer. The application then initializes the array of message descriptors so that each descriptor points to one of the message buffers.

Figure 5 illustrates the structure of a receiver managed portal. In this case, the message header information is recorded in the message buffer descriptor and the message bodies are separate from the message headers.

The array of buffer descriptors is managed as a circular queue. The Q-Kernel maintains a pointer to the next message buffer descriptor in the portal. When a message arrives, the kernel places the message in the message buffer identified by the next buffer descriptor and advances the buffer descriptor pointer to point to the next buffer descriptor.

In contrast to kernel managed portals, receiver managed portals require more explicit initialization on the part of the application programmer and offer very little flexibility. For example, all of the message buffers associated with a receiver managed portal must be the same length. To use this type of portal, the programmer must have a thorough understanding of the communication patterns exhibited by the application program.

Although they are more difficult to use, receiver managed portals can be more efficient in many applications. First, because the receiver can preallocate the message buffers, this strategy reduces the latency associated with message reception. Second, because the receiving application can control where messages are delivered, this strategy minimizes the need for the memory-memory copies that might be required when using a kernel managed portal. In many cases, the application programmer can arrange to have the buffer descriptors point directly to the locations where the data in the incoming messages need to be stored.

4.3 Sender Managed Portals

The memory associated with a sender managed portal is effectively managed by the processes that send messages to the portal. To create a sender managed portal, a process simply registers a block of its memory as the portal. Sending processes specify offsets into this block when they send messages. When a message arrives for a sender managed portal, the Q-Kernel transmits the body of the message to the portal memory at the specified offset.

The Q-Kernel does not record any structural information for a sender managed portal. It does not notify the process when messages are delivered to the portal. Moreover, it is possible for processes to overwrite messages sent by other processes. Sending processes must coordinate their use of sender managed portals. This includes notifying the process when a complete message has been delivered to the portal and avoiding overwrites in the portal memory.

Sender managed portals were designed to support parallel servers—collections of processes that provide shared resources for applications. For example, a parallel file server may be partitioned into several processes (perhaps one per disk). To read a block from a file, the application would start by allocating a block of its memory to hold the data. After registering this block of memory as a sender managed portal, the application would send a read request to one of the server processes. Different server processes could then fill in the appropriate portions of the portal memory block. Note that the application process does not need to know how the server is organized to make use of the resource provided by the server.

When server managed portals are used, the Q-Kernel does not record message header information for the incoming messages. Only the bodies of these messages are save in the portal space. Typically, the receiving application will set up another portal where it can be notified when all of the data values have been transmitted into the sender managed port.

4.4 Read Memory Portals

Read memory portals represent the converse of sender managed portals. As we have seen, application processes can use sender managed portals to provide a memory write operation for remote processes. Using read memory portals, application processes can provide a memory read operation for remote processes.

To establish a read memory portal, the application process registers a block of its memory as the portal. Request messages sent to a read memory portal specify an offset, a length, and a reply portal (on the requesting process). When the Q-Kernel receives a message for a read memory portal, it generates a response and sends it to the reply portal of the requesting process. The response message consists of the memory values starting from the specified offset in the portal and has the length specified in the request message.

Like sender managed portals, read memory portals were designed to support parallel servers. For example, to write a block of memory to a parallel file, the application starts by registering the data block as a read memory portal and sends a write request to a process in a server process group. The processes in the server process group can then send request messages to the read memory portal as they are able to consume blocks of the file.

This approach lets the server processes exercise a greater degree of control over their memory buffers. The parallel file server only needs to allocate a small amount of space for incoming requests. (The actual write request will be very short, specifying only the portal where the data are stored.) The server can then "pull" data from the application as it has buffer space available and at a rate that is compatible with the physical storage device.

4.5 The Portal Table

An application process may have several portals. Each portal has an associated portal descriptor that specifies the portal type and other important portal information. In the case of a receiver managed portal, the portal descriptor includes the address of the buffer descriptor array, an integer specifying the size of each message buffer, an integer specifying the number of buffers, and an index for the current message descriptor. For a sender managed portal, the portal descriptor only includes the starting address and length of the memory block associated with the portal.

All of the portal descriptors for an application are stored in an array called the portal table. Sending processes provide an index into the portal table of the receiving process when they send a message. When a message arrives, the Q-Kernel on the destination processor node first determines the destination process for the incoming message. The portal index provided by the sender is then used to determine which portal is to be used for receiving the message. Once the Q-Kernel has determined the target portal, it uses the portal descriptor to determine how the message should be mapped into the associated portal memory.

5 COMMUNICATION CONTROL

Because PUMA does not provide flow control in message transmission, it is possible that a malfunctioning or malicious process could flood the portals of an application process with invalid messages. In this way, an erroneous process can make it difficult (if not impossible) for other processes to send messages to the flooded process. The policies and mechanisms of PUMA do not overcome this difficulty, but do limit the scope of such attacks.

In PUMA, communication control is based on capabilities. All capabilities, including those used for communication, are managed and controlled by the PCT. Whenever an application process issues a communication request (i.e., when the application attempts to send a message), the Q-Kernel confirms that the process has the needed capability. The Q-Kernel uses the capability data structures constructed by the PTC to confirm that the process has the needed capabilities.

PUMA provides two types of capabilities that are used to control communication: group and portal capabilities. Group capabilities are associated with process group. A process holding a group capability can send a message to any portal on any process in the group. Portal capabilities identify a specific portal on a specific process. A process holding a portal capability can only send messages to the portal and process specified by the capability.

Roughly speaking, group capabilities are used for application-level communication whereas portal capabilities are used for communication with server processes. In application-level communication, portals are naturally used to distinguish different communication contexts. As such, group capabilities, which allow a process to send messages to any portal on any process in a restricted set of process groups, reflect common programming practice. In contrast, portal capabilities are analogous to file descriptors. After "opening" a resource, the application process can use the portal capability to send manipulation requests to the server process that provides the resource. Server processes can use the restrictive nature of portal capabilities to ensure that different applications (i.e., process groups) do not interfere with one another. In particular, the server process can associate different portals with different process groups to avoid difficulties associated with portal flooding.

Table 1. Types of Communication Capabilities in PUMA

	Any Portal	Specific Portal
Any process (in group)	Group capability	
Specific process		Portal capability

Table 1 summarizes the communication capabilities provided by PUMA. In examining this table, notice that PUMA does not provide capabilities that allow a process to send to a specific portal on any process in a process group or any portal on a specific process. Although these types of capabilities might be useful in some contexts, we did not feel that they were nearly as important as the group and portal capabilities currently provided.

5.1 Using Group Communication Capabilities

When an application process sends an application message, it identified the process group and rank id for the destination process along with the communication context and message buffer. Figure 6 illustrates the mappings performed by the Q-Kernel prior to transmitting an application-level message.

The group identifier supplied by the application process corresponds to a group communication capability that is stored in the address space of the PCT. Every group communication capability holds an array of pairs. The *ith* pair in this array identifies the processor node and the process control block (PCB) of the process with rank id *i*. (In this respect, every group communication

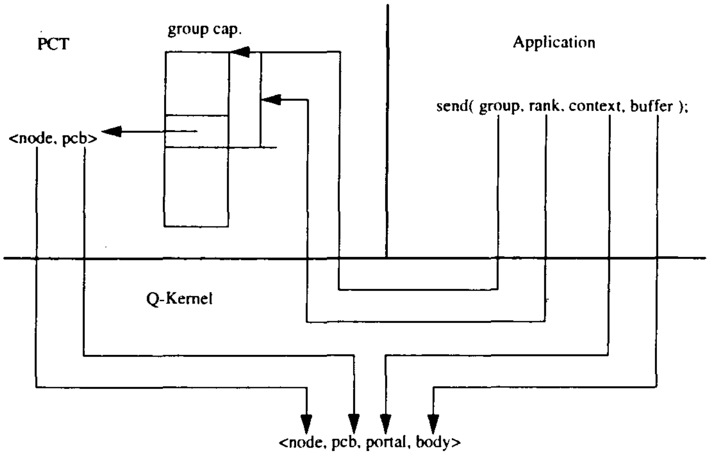
PCT
group cap.
Application
send(group, rank, context, buffer);
<node, pcb>
Q-Kernel
<node, pcb, portal, body>

FIGURE 6 Using a group communication capability.

capability acts as logical to physical map for the processes in the process group associated with the group communication capability.) As shown in Figure 6, the Q-Kernel uses the process rank id supplied by the application process as an index into the group communication capability.

5.2 Using Portal Communication Capabilities

When an application process sends a server message, it only needs to identify the object to be manipulated and the message buffer. Figure 7 illustrates the mappings performed by the Q-Kernel prior to transmitting a server message. In this case, object identifier supplied by the application process is mapped to a portal capability. Portal capabilities identify the processor node, PCB, and portal used in the actual transmission.

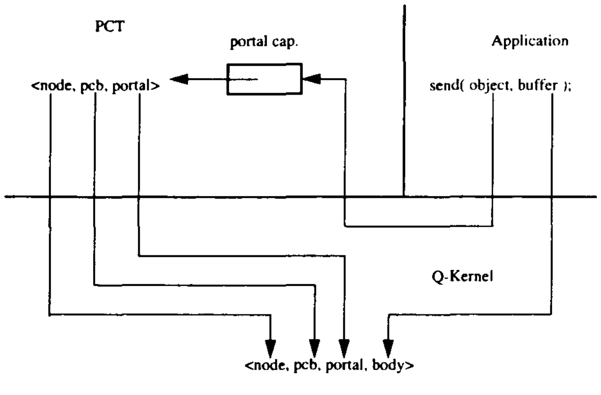

FIGURE 7 Using a portal communication capability.

5.3 Obtaining Communication Capabilities

When a process is created, it is given a group communication capability for its own process group. To obtain other group or portal communication capabilities, the process must make a request through the local PCT. Initially, the PCT acts as a name server to locate a server or process group with the desired attributes. This may involve communication with PCTs running on remote processor nodes.

If the goal of the search is to find a process group, the PCT obtains a copy of the group communication capability for the target process group and makes this capability available to the application process. If the goal of the search is to find a server process, the local PCT sends a connection request to the (possibly remote) server process. This request includes the information (e.g., user name) that the server needs to determine if the request is valid. If the server process accepts the request, it responds to the requesting PCT with identifier of the portal to be used by the application process. Using the response from the server process, the PCT on the application's processor node then constructs a portal capability and makes this capability available to the application process.

6 THE Q-KERNEL

The Q-Kernel is the most trusted level in the PUMA architecture. This is the only level in PUMA that has direct access to the address mapping and communication hardware. The Q-Kernel treats the communication network as a trusted and reliable resource. When a message arrives, the Q-Kernel assumes the communication reflects a message sent by a Q-Kernel running on another processor node. Because of this assumption, the Q-Kernel does not need to authenticate the source of the message or validate the contents of the message.

6.1 Data Structures

The Q-Kernel maintains two internal data structures: the process *context table* and the *outgoing message queue*. The Q-Kernel uses the process context table to switch between protection/privilege domains. In addition, entries in the context table have references to the per process data structures maintained by the PCT, e.g., the process control blocks.

The first entry in the context table is associated with the PCT. This entry is initialized when the Q-Kernel is loaded. The remaining entries are established by the PCT whenever it loads an application or server process. The number of entries in the context table, and hence, the number of processes per processor node, is fixed when the Q-Kernel is loaded.

The outgoing message queue has an entry for every message that has been registered in the Q-Kernel but not transmitted. This may include messages from the PCT, messages from application and server processes, and messages generated in response to messages sent to read memory portals.

To maintain a fixed number of entries in the outgoing message queue, the Q-Kernel establishes an upper bound on the number of entries that any process can have in the queue at any time. If a process exceeds this upper bound, the Q-Kernel rejects transmission requests from the application (including responses to its read memory portals) until the application has more queue entries available.

Note that the Q-Kernel does not block application processes. However, an application process can arrange to have the PCT block its execution and have the Q-Kernel generate a signal when there are queue entries available.

6.2 Entry Points

The Q-Kernel can be activated by a user-level call (a Q-Kernel entry point), an exception (e.g., divide by zero), an interrupt associated with the communication hardware, or a timer interrupt. We begin by considering the user-level entry provided by the Q-Kernel.

The Q-Kernel provides seven entry points. Two are associated with message transmission (send_app_msg and send_server_msg). The third is used to restore an execution context (run_context). The fourth is used to suspend the execution of an application process (quit_quantum). The fifth and sixth are used to establish protection/privilege domains for application processes (create_context and extend_context). The seventh is used to establish the execution context for the PCT (set_PCT).

Three of the Q-Kernel entry points (create_context, extend_context, and set_PCT) are restricted to the PCT, and cannot be successfully invoked by an application or server process. Two other entry points (send_server_msg and quit_quantum) are only useful for application and server processes. The two remaining entry points (send_user_msg and run_context) can be used by application processes as well as the PCT.

Figure 8 summarizes the Q-Kernel entry points. We have already discussed the activities that the kernel performs in response to a call to the *send_app_msg* and *send_server_msg* entries. In the remainder of this section, we consider the activities that the Q-Kernel performs in response to the remaining entries.

The *run_context* entry restores an execution context. This entry is used by the PCT to resume the execution of an application process, i.e., to dispatch the process. Application processes can also use the run_context entry to resume execution of a saved context after handling a signal. (In many machines, the application process will be able to restore a saved context without invoking

```
send_app_msg( capability group, int process, portal_id portal, int offset,
        void *buf, size_t len, int *flag );

send_server_msg( capability descriptor, void *buf, size_t len, int *flag );

run_context( int context_num, saved_state *proc_state, int time_quantum );

quit_quantum( void *mail_box );

create_context( cntx_index context );

extend_context( cntxt_index context, address_map map );

set_PCT( address thread_start );
```

FIGURE 8 Prototypes for the Q-Kernel entry points.

the kernel; however, in some machines an application process cannot easily restore the entire processor status word.)

For the most part, the Q-Kernel does not distinguish between the PCT and an application process when it is handling a *run_context* invocation. However, it does allow the PCT to establish a time quantum for the execution of an application process. If this time quantum expires, the application process will be interrupted and the Q-Kernel will transfer control back to the PCT.

An application process can use the *quit_quantum* entry to suspend its activities. This capability can be used to wait until a message arrives and, as we will discuss, to request a service from the PCT. When an application process invokes the quit_quantum entry point, the Q-Kernel saves the context of the process and runs the PCT.

The last three entry points are restricted to the PCT and cannot be called by application or server processes. Two of these entries, *create_context* and *extend_context*, are used to establish the address space for an application or server process. The PCT calls the create_context entry to create an initial (empty) address map for an application process. After creating the initial map, the PCT can extend the application's map by adding portions of its memory to the application's map. Whenever the PCT extends the address map for an application context, the Q-Kernel validates that the memory used in extending the map is owned by the PCT (and not the Q-Kernel).

The final entry point, *set_PCT*, is used to establish the restart address for the PCT. When PUMA is first loaded, the Q-Kernel builds a complete execution context for the PCT and transfers control to the initial start address for the PCT. After initializing its data structures, the PCT registers its continuation address with the Q-Kernel using the set_PCT entry. Subsequent entries into the PCT use the continuation address instead of the initial start address.

6.3 Exceptions

When an application process encounters an exception during its execution (e.g., divide by zero, address fault, etc.), the Q-Kernel records the pertinent information, sets a signal bit in the PCB (maintained by the PCT), and invokes the PCT. The PCT can then handle the exception by terminating the process or transferring control to the application's signal handler.

6.4 Communication Interrupts

The Q-Kernel handles two types of communication interrupts: transmit complete and message arrival. When the Q-Kernel determines that an outgoing transmission was completed, it notifies the sending process that the transmission has been completed (by incrementing the flag and setting a signal bit in the sender's PCB). The Q-Kernel also notes that the sending process has a free queue entry and initiates the transmission of the next outgoing message, if there is one.

When the Q-Kernel receives an interrupt for an arriving message, it uses the process index in the incoming message as an index into its context table. This entry identifies the PCB for the destination process. The PCB identifies the portal table that, in turn, identifies the destination portal. Before it initiates the message receipt, the Q-Kernel verifies that the group identifier in the message matches the group identifier of the receiving process and that the message will not violate the domain constraints for the destination process portal. When the message body has been received, the Q-Kernel updates the destination portal's descriptor and sets a bit in the array of pending signals for the process.

6.5 Timer Interrupts

Whenever the Q-Kernel receives a timer interrupt, it means that the current application has exceeded its time quantum. In this case, the Q-Kernel simply runs the PCT.

7 THE PCT

As we have discussed, the PCT provides many of the services typically associated with an operating system, e.g., process creation, process scheduling, and connection to other services. The PCT is run whenever an application process completes its allocated quantum, either because of a timer interrupt or because the application invoked the quit_quantum entry. It is worth noting that the PCT is not run after every entry into the Q-Kernel.

7.1 An Overview of the PCT

The PCT begins its execution by checking for requests from the application process whose execution was suspended. When an application process needs a PCT service, it prepares a PCT request and places the request in a mailbox structure. In most cases, an application process will prepare a single request, place the request in its mailbox, and voluntarily relinquish the CPU using the quit_quantum entry into the Q-Kernel. An application can prepare several requests and link them into its mailbox. The PCT only examines the application's mailbox after the application suspended its activities using a quit_quantum request. In particular, the PCT will not examine an application's mailbox if the application's execution was suspended due to a timer interrupt. This strategy avoids race conditions associated with the mailbox structure.

After checking the mailbox of the suspended process, the PCT examines its portals for messages from other PCTs. PCTs send messages to other PCTs when they need information about the resources and services provided by another node. This type of information is needed when an application initiates a server connection or spawns a process.

After handling requests from other PCTs, the PCT tries to find a runable application or server process. If the PCT does not find a runable process, it enters a loop in which it responds to new requests from other PCTs and checks for a runable process. When it finds a runable process, the PCT selects one of the runable processes as the next process to be executed. The PCT always completes its activities by invoking the Q-Kernel run_context entry, thus dispatching the selected process.

7.2 Signals

Each process has a vector of pending signals. In addition, each process maintains a vector of blocked signals, a vector of ignored signals, and a signal handler function. When a process has a pending signal that is neither blocked nor ignored, the PCT notes that the process has an outstanding signal. Any process with an outstanding signal is runable.

Before the PCT runs a process, it checks to see if the process has an outstanding signal. If it has an outstanding signal, the PCT copies the saved context into the stack for the process and establishes the signal handler as the current execution context. This makes it relatively easy to stack signal handlers and passes the burden for storing stacked contexts back to the application or server process.

7.3 Blocking

To block its execution, an application or server process sets its execution state to blocked and invokes the quit_quantum entry provided by the Q-Kernel. When the PCT looks for runable processes, it notes that the process is not runable and the process will remain blocked until it receives a signal.

8 RELATED WORK

Many researchers have recognized the need to minimize memory copies during communication [7]. Mach, for example, uses page mapping to avoid memory copies [8]. Incoming messages are held in kernel pages until requested by an application. When an application requests a message, the kernel maps the pages holding the requested message into the application's address space. This approach can avoid the costs associated with memory copies and presents the application programmer with simple IPC semantics. In developing PUMA, we had two concerns with the memory mapping approach. First, some MP machines (e.g., the nCUBE2) do not provide adequate address mapping hardware to support this approach. Second, to avoid memory copies, applications must receive messages in buffers that are aligned on page boundaries. This requirement may waste physical memory and impose a significant burden on the organization of the data structures used in an MP application.

The "active messages" approach avoids memory copies by invoking a user-level message handler whenever a message arrives from the communication network [9]. Because the user-level handler controls the placement of incoming messages, this approach should minimize the need to copy messages. However, because the user-level message handler has direct access to the communication hardware, this approach may not be appropriate when several applications share the resources provided by an MP machine.

9 RESULTS

In January 1993 we completed a preliminary implementation of PUMA for the nCUBE2. This implementation achieves message throughput rates as high as 2.17 Mbytes per second per channel, or 98% of the channel capacity. Our current message latencies (application level to application level) are approximately 110 microseconds. With turning we expect that we can reduce the latency by 25% or more.

In May, we began porting PUMA to the Intel Paragon. We completed an initial implementation in August 1993. This implementation achieves message throughput rates as high as 165 Mbytes per second per channel, or 94% of the channel capacity. Every processor node on the Paragon provides two i860 processors. Message latencies in our Paragon implementation are approximately 50 microseconds when we use a single processor per node and as low as 30 microseconds when we use the second processor to handle all message traffic.

ACKNOWLEDGMENTS

We are grateful to the members of the original SUNMOS team. Without their efforts, we would not have been in a position to undertake the development of PUMA. We would also like to acknowledge the efforts of the other members of the PUMA team, including Kenneth Ingham, Clint Kaul, Michael Levenhagen, and Francisco Reverbel.

REFERENCES

[1] R. van Renesse and A. S. Tanenbaum. *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures.* 1992, pp. 1–10.

[2] M. Rozier, et al., "The chorus distributed operating system," *Comput. Systems*, Vol. 1, 1988.

[2] J. Alemany and E. W. Felten. *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Cqmputing.* 1992, pp. 125–234.

[3] M. J. Accetta, et al., *Proceedings of the Summer 1986 USENIX Conference.* 1986, pp. 93–113.

[4] R. Zajcew, et al., *Proceedings of the Winter 1993 USENIX Conference.* 1993, pp. 449–468.

[5] D. R. Cheriton, "The V Kernel: A software base for distributed systems," *IEEE Software*, Vol. 1, pp. 19–42, 1984.

[6] D. R. Cheriton. "The V distributed system." *Communications ACM*, 1988.

[7] C. M. Burns, R. H. Kuhn, and E. J. Werme, *Proceedings of Supercomputing.* 1992, pp. 760–769.

[8] J. S. Barrera III, *Proceedings of the USENIX Mach Symposium.* 1991.

[8] A. B. Maccabe and S. R. Wheat, "The PUMA architecture: AN overview." Sandia National Laboratories Technical Report SAND93-1372.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, *Proceedings of the 19th International Symposium on Computer Architecture.* May 1992.