SANDIA REPORT

SAND98–2221
Unlimited Release
Printed October 1998

Differences Between Distributed and Parallel Systems

Rolf Riesen, Ron Brightwell, Arthur B. Maccabe

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND98-2221
Unlimited Release
Printed October 1998

Differences Between Distributed and Parallel Systems

Rolf Riesen and Ron Brightwell
Computational Sciences, Computer Sciences, and Mathematics Center
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Arthur B. Maccabe
Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131

Abstract

Distributed systems have been studied for twenty years and are now coming into wider use as fast networks and powerful workstations become more readily available. In many respects a massively parallel computer resembles a network of workstations and it is tempting to port a distributed operating system to such a machine. However, there are significant differences between these two environments and a parallel operating system is needed to get the best performance out of a massively parallel system.

This report characterizes the differences between distributed systems, networks of workstations, and massively parallel systems and analyzes the impact of these differences on operating system design.

In the second part of the report, we introduce Puma, an operating system specifically developed for massively parallel systems. We describe Puma portals, the basic building blocks for message passing paradigms implemented on top of Puma, and show how the differences observed in the first part of the report have influenced the design and implementation of Puma.

Acknowledgment

Operating systems work is seldom carried out by a single person. We would like to thank the other members of the Puma team for their help, insights, and friendship. Specifically, David van Dresser, Lee Ann Fisk, Chu Jong, and Mack Stallcup have contributed to this report with their ideas, designs, and implementations. We also would like to thank Lance Shuler, David Greenberg, George Hartogensis, and the anonymous reviewers who assisted by reading our manuscript and made polite suggestions to improve it. Most of all we would like to thank our users, who have been willing to test our systems even in early stages of development. Their feedback helped in the design of Puma and is much appreciated.

Contents

Figures

Tables

Preface

This technical report, written in late 1996, presents an analysis of the differences between distributed and parallel systems. While many of the research projects referenced in this report have concluded and many of the examples and results are obsolete, the fundamental basis of this report remains intact: unique characteristics of each system need to be exploited to meet the overall goals of the platform.

Differences Between Distributed and Parallel Systems

1 Introduction

Distributed (computing) systems have been investigated for about twenty years. In 1985 Andrew Tanenbaum and Robert van Renesse offered the following definition for an operating system controlling a distributed environment:

> A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs) [30].

No existing system has completely achieved the functionality mandated by this definition. However, research in distributed systems and networks of workstations (NOWs) has produced a number of useful results in the areas of algorithms, languages, and resource management.

With the advent of massively parallel (MP) systems, it seems straightforward to use the technology developed for distributed systems and apply it to the latest generation of supercomputers. After all, the individual nodes of an MP system are similar to workstations lacking only an attached keyboard, monitor, and possibly disk. Furthermore, many MP and large distributed systems compute similar problems.

Attempts to treat an MP system as if it were the same as a distributed system have produced disappointing results. Performance has often not been as high as expected and has been well below the limits imposed by the hardware.

There are fundamental differences in the structure, usage, and performance of MP and distributed systems. An operating system for MP machines must take advantage of these differences and provide a different set of services to the applications than an operating system designed for a distributed environment.

This report explores the observable differences between a distributed system and a modern MP machine and tries to separate the differences that are artifacts of historical development from the ones that are fundamental. As technology improves, differences in characteristics and performance will get smaller and eventually disappear. However, some differences will not disappear. For this reason, an operating system for an MP system cannot be the same as one for a distributed system.

This report is divided into two parts. The first part discusses the differences between distributed and MP systems. The second part gives an overview of Puma, an operating system specifically developed for an MP system.

2 System Environment

In this section, we consider differences in the environment presented to users and applications. We examine how resources in the system are managed, how the systems are used, and what system behavior users and applications see and expect.

A distributed system usually consists of a set of workstations. Idle resources during the day, and especially during the night, led to the desire to treat individual workstations as part of a larger integrated system. In contrast, an MP system is purchased and tailored for high-performance parallel applications. These applications use all available resources. The different reasons for the existence of a particular MP or distributed system lead to a set of observable differences that we identify in this section.

Differences can be observed in a number of areas. The types of resources, their management, and the functionality offered by the systems are different between distributed and MP environments. The number of processes per node as well as the number of users on each node is also different. Granularity of parallelism is not the same, and the problem of gang scheduling is handled differently in each environment. The two types of systems also locate services and peripheral devices in different areas of the machine.

Resources: The primary goal of a workstation is to provide a user interface for a computing environment. The resources present in a workstation are determined by the need to provide a highly interactive user

interface. Importantly, resource utilization is not the primary influence in the acquisition of additional resources for a workstation. Thus, idle resources will be available. The desire to utilize these idle resources has led to the idea of combining workstations into a distributed system or a NOW [2]. The resources of a workstation (memory, processor, and disks) can be assigned to other applications while the user is not making full use of them. It is the task of a distributed operating system to locate and manage available resources.

In an MP environment, utilization drives the acquisition of resources. A new MP machine is carefully tailored to the application with the largest demands on resources. When this application is running, no idle resources are available for other applications. For distributed systems, resources must be accessible as efficiently as possible.

Resource management: Various programming models have been proposed for distributed systems. All have the common goal of maintaining the parallelism and the structure of the underlying system hidden from the users. Ideally, an application should take advantage of as available resources without the need to write or run the application in a different manner than on a single workstation.

An often cited example is parallel *make* which can distribute the compilation of individual modules onto several nodes.

In an MP system, the primary goal is to minimize the turn-around time for an application. To achieve this, a set of nodes is assigned to a specific user for the duration of an application run.

High-performance MP applications, do resource management. Load balancing is either inherent in the algorithms used, or applications make use of libraries which dynamically shift the computational load. Information about the topology and the location of neighboring nodes is frequently used to minimize network congestion and latency.

Functionality: Distributed systems have the conflicting goals of providing full workstation functionality and a pool of available resources to run parallel programs efficiently. There is functional overhead on nodes which have been dedicated to parallel applications. The compute nodes in an MP system only need the functionality required to run parallel applications. Many of the time and space consuming features required in a general purpose workstation can be omitted in a parallel operating system.

Users per node: The nodes in a distributed system are full featured workstations with a monitor, keyboard, and local disk storage. The workstations in most distributed systems must serve and respond to one or more interactive users. In addition, the workstations may be required to participate in distributed computations[1]. Typically there are one or more users per node in a distributed system.

In an MP system there are many nodes dedicated to a single user. The individual nodes and the operating system do not directly support interactive users. (This does not preclude interactive parallel programs; interaction is with the entire program through messages sent to the nodes.)

Processes per node: Another distinguishing feature between distributed and MP systems is the number of processes per node. A single workstation in a distributed system must manage tens, sometimes hundreds, of processes. These processes include daemons (e.g. networking daemons, print spool managers, cron jobs, etc.), a window system server and its clients, and background processes.

Most of the services provided by these processes are not needed by parallel applications, and most do not need to be replicated on every node in the system. Therefore, a parallel operating system can be optimized for a small and fixed number of processes per node.

Parallelism: We have mentioned parallel *make* as an application well suited for a distributed systems. This application exploits parallelism at the program level, where several, independent, programs can be distributed among a set of available nodes.

MP applications exploit parallelism at a much finer granularity. Applications use a large set of nodes and communicate more frequently than distributed programs.

Gang scheduling: The individual processes of a parallel application in a distributed system are under the scheduling control of the local operating system. Different workstation configurations and current utilization determine the frequency and duration of the time slices allocated to each process. It is quite unlikely that the processes of a parallel application will actually run in parallel (Figure 1). This has a severe impact on performance. Large data transfers require the immediate consumption of data on the receiving side. Otherwise, data buffers will fill and the sending process must be blocked. Bursts of small requests

[1] Amoeba [31], which has dedicated compute servers, is an exception.

also require immediate attention. If the individual processes in a parallel application are scheduled to run at different times, message latency will rapidly increase, bandwidth will drop, and overall performance will suffer.

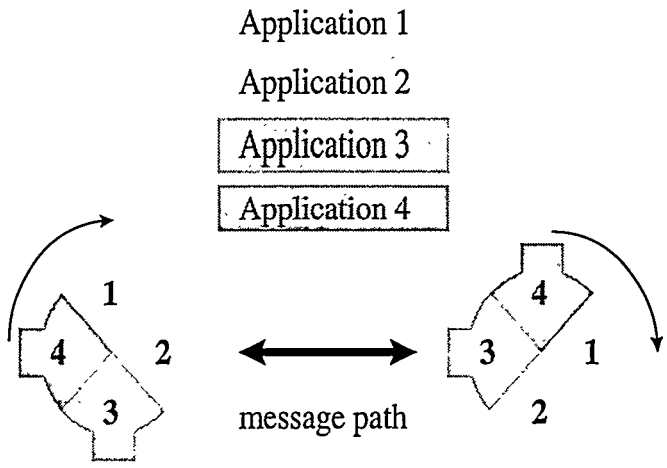

Figure 1: **Gang Scheduling:** The two sprocket wheels represent the process scheduling on two separate nodes. Application 2 on the left and application 3 on the right are active. They could exchange data immediately, since both are active and have access to the communication network. However, application 2 on the left must wait for the wheel on the right to turn until application 2 on that side gains access to the communication channel, before a reply to a message can be sent. Gang scheduling would synchronize the two wheels so that the processes belonging to the same application are always lined up with each other.

Many applications written for MP machines are self-synchronizing. Data exchanges among nodes happen at regular intervals, and nodes cannot proceed with computation until results of the last time-step have been received. If the individual processes do not run concurrently, resources will be wasted. Additional buffer space and context switches are necessary to allow the application to progress.

Location of services: The individual workstations in a distributed system often have local disks. Files accessed by a parallel application might be on a network file server, or might be on local disks. Other shared resources, such as access to an external network, printers, and other external devices, are present on some workstations, but not others. A distributed operating system creates the illusion that resources and services are all available locally.

The nodes in an MP system are usually uniform and no additional services or resources are provided locally. Therefore, access is always remote, and the request is satisfied remotely.

In this section, we have listed differences between distributed and MP systems observed by users and applications. In most cases, these differences are due to differences in the node architecture and the network connecting the individual nodes. We will examine these aspects of distributed and MP systems in the next two sections.

A recent trend has been to buy individual workstations, connect them with a high-speed network such as fast Ethernet, ATM, or Myrinet, and use these components as a dedicated NOW. In this case, many characteristics of an MP machine apply, and the operating system controlling the NOW should have the characteristics of a parallel operating system controlling an MP system.

3 Node Architecture

Our second characterization of differences between distributed and MP systems is based on dissimilarities in the node architecture. In this section we analyze the characteristics of compute nodes in an MP system and compare them to the workstation architectures found in distributed systems.

There are differences in the type and number of devices attached to a node. The nodes in a distributed system are often heterogeneous, and the network interface is integrated differently.

Node devices: While the nodes in an MP system share many of the characteristics of a modern workstation (e.g. CPU, memory, etc.), there are differences that must be considered. The nodes in a distributed system are full-featured workstations with keyboards, monitors, disks, and other peripheral I/O devices. Most nodes in an MP system have no peripherals other than a network interface. All requests and data arrives at the node in the form of messages through the network interface.

Node homogeneity: Distributed systems are much less homogeneous than MP machines. The individual workstations often have CPUs with different performance ratings or different architectures. There are different amounts of physical memory, possibly different operating systems, and varying configurations (number and size of peripherals such as disks, printers, tape drives, etc.) to contend with. The nodes in the compute partition of an MP system may have different amounts of memory, but are otherwise very similar. An exception are I/O nodes, which look like regular compute nodes but have additional interfaces to peripherals.

A parallel operating system should take advantage of this homogeneity. For example, system software can assume that the data representation does not change from one node to another and omit the functionality to convert from one format into another.

Network interface: The network interface in an MP node is usually more integrated into the node architecture than in a typical workstation (Figure 2). Some MP systems make it possible to access the network interface from user level. Most provide direct memory access (DMA) to local node memory (if not the capability to transfer data to and from the cache or even the registers of the CPU). This tight integration with the CPU and memory subsystem on the node allows for low-latency and high-bandwidth access to the network(see [18], [23], or [17] for examples).

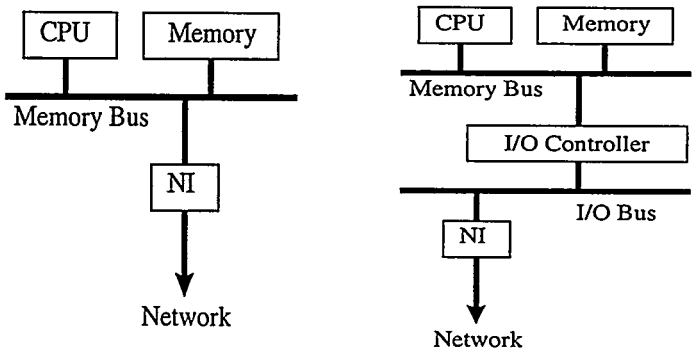

Figure 2: **Network interface (NI) integration:** On modern MP systems the network interface is either integrated into the CPU or is tightly coupled with the memory and CPU on a node, as shown on the left. Most workstations have an I/O controller, and the network interface is another device on the I/O bus. Often, this is a relatively low speed bus designed for high latency devices, such as disk, modems, and printers.

In contrast to the tight integration in MP nodes, the network interface in a typical workstation is another peripheral device, often without intelligence and with no direct access to local memory. The CPU must copy data between the buffers of the network interface and local memory, or transmit data over a relatively slow I/O bus.

A tight integration is necessary to avoid memory copies. The ratio of network bandwidth to memory copy bandwidth is increasing as networks become faster. One or more memory copies per transfer is no longer acceptable. In the future, approaches designed to avoid memory copies, such as Puma portals, will

be crucial to achieve high bandwidth and low latencies.

Figure 3 shows the performance of a memory copy on successive implementations of the Sun SPARCstation architecture and compares it to the network bandwidth achievable in an MP system. Compared to workstation networks such as Ethernet ($\sim 1MB/s$), Fast Ethernet ($\sim 10MB/s$), ATM (OC-12 $\sim 80MB/s$), and Myrinet ($\sim 75MB/s$), memory copy speeds of modern workstations (for example a SPARC Ultra model 170) are good enough and allow one or two copies to be made during a message transfer. With the network bandwidth achievable in an MP system (Intel Paragon $\sim 155MB/s$), a single memory copy is disastrous.

For example, in a system with a $160MB/s$ network and the necessity to copy network data from a buffer to memory (also at $160MB/s$), the overall bandwidth can be at most $80MB/s$. Messages could be packetized, and the memory copy of one packet can be done at the same time as the next packet is arriving. This strategy does not help much. Shorter packets have lower bandwidth on the network due to start-up overhead. Also, the memory bus must be shared between the memory copy and the reception of the incoming packet.

A parallel operating system must be highly tuned for message passing and must avoid memory copies.

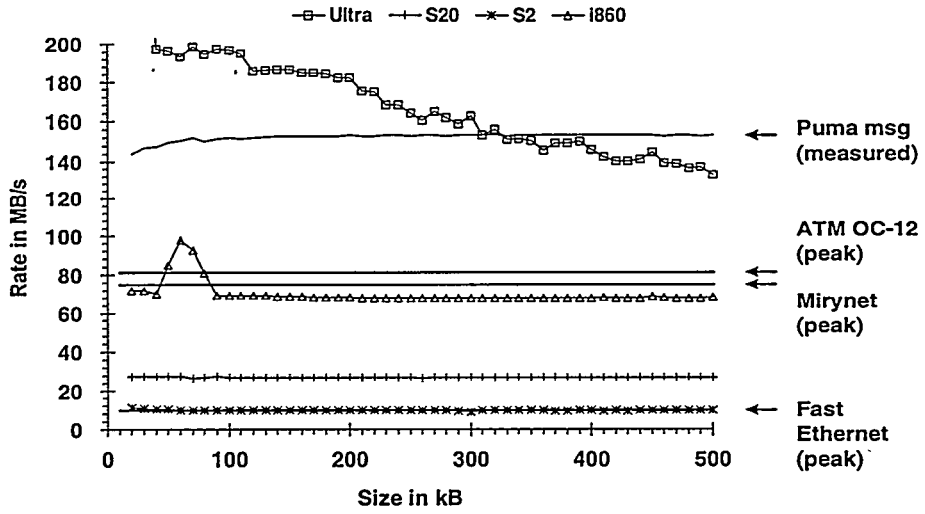

Figure 3: **Bandwidth:** Memory copy speeds of successive generations of Sun SPARCstations are keeping up with workstation network speeds. In an MP system, memory copies are disastrous during a message transfer, due to the high bandwidth of MP networks. Performance was measured on lightly loaded SPARC systems and pre-release Puma (i860). Measured message passing bandwidth is from pre-release Puma on Intel Paragon. ATM, Myrinet, and Fast Ethernet numbers are peak for technology.

4 Node Interconnect

In this section, we examine the network itself. The network connecting the individual workstations in a distributed system, and the communication fabric shared by the individual nodes of an MP system, exhibit differences in trust, bisection bandwidth, topology, broadcast ability, network transparency, prevailing communication paradigm, IPC performance, and configuration.

Network trust and reliability: Distributed systems can span a campus, a country, or whole continents. The network connecting the individual pieces crosses administrative domains and often consists of several different physical media (e.g., example public telephone network, satellite connections, radio links, Ethernet, etc.)

The network of an MP system is typically contained in a single cabinet or in a small collection of cabinets located in a single room. The distance between two adjacent nodes is only a few centimeters. Hardware

cyclic redundancy checks (CRC) and parity error detection keep the probabilities for undetected errors very low. In addition, the network can easily be protected from physical intrusion, since it is confined to a single room.

The kernels that compose a parallel operating system can trust each other and ensure the integrity of system information in message headers. System information, such as source and destination address, message length field, and process identifiers, is needed to route messages to the appropriate destination and protect one application from another. The ability to trust each other reduces the amount of checking the kernels must do and reduces the amount of system information (overhead) in each message. This leads to lower latencies and higher bandwidths.

Bisection bandwidth: Bisection (or cross-section) bandwidth is an important measure in MP systems. It is the bandwidth through an imaginary plane, dividing the nodes in a system into two, equal sized, sets (Figure 4). In a well balanced MP system this number is many times the bandwidth of a single link. On a bus or ring, the bisection bandwidth is usually one or two times the bandwidth of a single link between two nodes. Generally, the achievable bandwidth in an MP system is an order of magnitude higher than in distributed systems. The individual network links are faster, the bisection bandwidth is much higher, since the network interface is more tightly integrated into the node.

A large bisection bandwidth connotes more independent pairwise paths between nodes. This lowers congestion on the network and improves application communication performance.

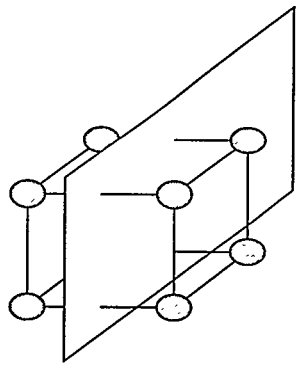

Figure 4: **Bisection bandwidth:** The nodes in a system are divided into two equal sized sets by an imaginary plane. The sum of the bandwidth of the communication links crossing the plane is the bisection bandwidth. The plane is placed so that the number of links crossing it is minimal and the number of nodes on each side is equal.

Topology: The nodes of a distributed system are usually connected with a bus-like network, such as Ethernet, or a ring, which do not scale well. As more nodes are added to the system, bandwidth between any two nodes is lowered because the network capacity must be shared with the new nodes. Adding gateways or star routers to keep local traffic in a subnet does not solve the scalability problem because the bisection bandwidth is not increased.

The nodes of a scalable MP system are arranged in a hypercube, 2-D or 3-D mesh or torus, a fat tree, or some other scalable topology. A parallel operating system needs to expose the topology to the application. A well tuned library can take advantage of the knowledge of who the nearest neighbors are. Using non-blocking message transfer operations, a library or application can also achieve higher throughput by saturating all links that lead to a node.

Hardware broadcast: Distributed systems connected with an Ethernet or other bus-like networks often have broadcast and multicast abilities. A message sent over the bus can be read by any node on the same network. A distributed operating system makes the decision whether the message should be delivered to local processes or not at the receiving node. In contrast, modern MP machines use wormhole routing and messages are only delivered to a single node[2].

[2]Some machines, such as the Intel Paragon, provide a hardware broadcast. However, it is restricted to the nodes in a rectangular region, and is seldom used because of the possibility of deadlock when it overlaps with another broadcast.

An operating system or an application running on an MP system that assumes a hardware broadcast is available, will not perform well in a large system. A parallel operating system must expose the network to the application level and use algorithms that are tuned to a particular topology, to do broadcast and other global operations well.

Network transparency: The goal of a truly distributed system is to be network transparent. Individual processes in a distributed application should not need to know where other processes are located. In fact, this information is often purposely hidden. Conversely, a well-written parallel application needs this information to place processes that communicate frequently onto nodes which are physically adjacent on the network.

Communication paradigm: Remote procedure calls (RPC) are the dominant form of information exchange in distributed systems. MP systems use explicit message passing. Some MP systems support distributed shared memory (DSM) in hardware and provide the illusion of a single large memory without the need for message passing. To maintain scalability, even DSM MP systems use small messages at the lowest hardware level.

Invoking a remote procedure (handler) for each message transfer incurs overhead that should be avoided when a simple data transfer is sufficient. This is especially true for MP systems with high network bandwidth. Any overhead will decrease the achievable bandwidth and impact latency.

IPC performance: Interprocess communication (IPC) performance is one or two orders of magnitude lower in distributed systems than in MP systems [19, 3]. This gap is closing as the network interfaces in distributed systems get more closely integrated into the node (workstation) architecture and the network performance of distributed systems is increasing [32].

As long as the difference remains significant, operating systems and applications must take them into consideration.

System Configuration: MP systems are more static than distributed systems. For any given system, it is known at boot time how many nodes there are, what the configuration of the peripheral devices is, and how the machine is going to be partitioned into service, compute, and I/O regions. In a distributed system, the amount of available resources fluctuates as workstations become loaded or idle.

A distributed operating system must have the ability to tolerate changes in the configuration. An operating system specifically written for an MP architecture can take advantage of the static configuration and omit many services required in a distributed system. A parallel operating system should be fault tolerant and be able to handle node failures and faulty network links. The overhead required for these operations is smaller than in a distributed operating system, since the maximum configuration of the MP machine is known at boot time, and only faulty components must be removed from service. Even if hot-swapability is available, the system does not grow beyond the configuration established at boot time.

5 Convergence of Parallel and Distributed Systems

Table 1 enumerates the differences in the order they were presented in the previous three sections. In Section 2 we looked at differences that can be observed by users and applications. The reason that these differences exist are mostly due to differences in the node architecture (Section 3) and the interconnect (Section 4).

The existing dissimilarities make it necessary to design operating systems specifically for MP systems. If, at some time in the future, MP and distributed systems sufficiently converge, then it will no longer be necessary to have two different types of operating systems.

The location of devices in an MP system will remain confined to a few service nodes. Cost and scalability prevent the duplication of devices such as disks and Ethernet interfaces on each node in the compute partition. Similarly, it does not make sense to have a monitor or keyboard attached to each node in a system consisting of thousands of nodes.

Since the goal of a distributed system is to make use of idle resources, these systems must deal with non-homogeneous nodes. Additional nodes will not usually be excluded from a distributed system because they have a different CPU type. Distributed systems are designed to handle this situation and any additional resource can be utilized. This may not be the case for a dedicated NOW. These systems lay somewhere between MP and distributed systems. As in a distributed system, they connect a number of workstations to form a single computer. However, certain precautions, such as making sure that all the workstations are the

Table 1: Summary of Distinguishing Features

Criteria	MP System	Distributed System
Resources	Dedicated	Utilize idle resources
Resource management	By the application	By the system
Perf. / Functionality ratio	High	Low
Users per node ratio	Low	High
Processes per node	Few	Many
Parallelism	Fine grained	Coarse grained
Gang scheduling	Yes	No
Location of services	Remote	Transparent
Node devices	No or few I/O devices	Full featured
Node homogeneity	Homogeneous	Heterogeneous
Network interface	Integral part of node	Separate device
Trusted network	Yes	No
Bisection bandwidth	High	Low
Scalable topology	Yes	No
Hardware broadcast	No	Yes
Network transparent	No	Yes
Communication paradigm	Message passing	RPC
IPC performance	High	Low
Configuration	Static	Dynamic

same, allow these kind of NOWs to support applications that ordinarily require an MP system to be solved.

More frequently, workstations are integrating the network interface to access memory directly. With an increasing demand for NOWs and better network performance, all workstation vendors may soon choose to offer this option.

So far, we have no convergence in the location of devices and the homogeneity of nodes (with the possible exception of dedicated NOWs). There is at least an indication of convergence as far as the integration of the network interface is concerned. Now, let us consider the differences in the interconnect.

As distributed systems grow and eventually span the globe, there is no hope of ever being able to trust the network. Exactly the opposite is true for MP systems, since the same processing power can be put into smaller and smaller spaces. Physical protection of the backplane (network) is more easily achieved, since the system can be confined to a small room or even a desktop. Again, NOWs lay somewhere in the middle between these extremes. Most of the NOWs consist of tens to hundreds of workstations that can be housed in a single, or a few adjacent, rooms. Physical protection of the network is feasible, though harder to achieve than in an MP system.

Bisection bandwidth is largely a function of topology and the networks used in distributed systems are not scalable to thousands of nodes. This is not likely to change in the foreseeable future, yet a high bisection bandwidth is crucial for many high-performance applications. This is one of the key differences between distributed and MP systems.

Hardware broadcast ability is disappearing from distributed systems due to the emergence of smart hubs in 10baseT Ethernets and the push towards networks such as ATM over fiber-optics, and Myrinet which are more point-to-point than the bus-like Ethernet.

Network transparency is a main goal in distributed systems and has advantages such as the ability to relocate services or substitute failed nodes. In order to get the highest possible performance, MP applications need to take advantage of the network topology and node allocation. This means the network cannot be transparent in an MP system.

Message passing systems, such as MPI [15] and PVM, are being used on NOWs. At the same time, certain forms of RPC, for example active messages [34], are now being used in MP systems. Some convergence is taking place.

IPC performance is a function of software overhead and network interface performance. As the network interfaces get more closely integrated and low-overhead protocols, such as U-net [33] and fast messages (FM) [24], start taking advantage of these interfaces, the distinction between MP and distributed systems will not remain a distinguishing factor.

Finally, the environment of an MP system will, by its nature of being a single machine, remain static in the sense that it is known at boot time what the maximum configuration can be. Distributed systems, as they become larger, spanning continents and even the entire globe, are by their nature very dynamic.

Table 2 summarizes our beliefs whether MP and distributed systems will converge and the observable differences will disappear. We have seen that for some differences, the answer is slightly different for NOWs, especially dedicated NOWs that are built for the purpose of high-performance computing.

Table 2: Convergence between MP and distributed systems and (dedicated) NOWs

Criteria	Convergence?	(Dedicated) NOW?
Node devices	No	No
Node homogeneity	No	Some
Network interface	Probably	Yes
Trusted network	No	Maybe
Bisection bandwidth	No	No
Scalable topology	No	No
Hardware broadcast	Yes	Yes
Network transparent	No	No
Communication paradigm	Some	Some
IPC performance	Probably	Yes
Configuration	No	No

6 Puma Overview

Puma [35, 27] is an operating system specifically designed for scalable, high-performance applications in an MP environment. Today, most of the applications running under Puma and its predecessor, SUNMOS, are scientific applications. However, the design of Puma has been influenced by other types of applications as well. Database engines, video servers, and other applications, such as simulations of economic models, should run well under Puma. An effort currently under way, will make Puma available in secure, real-time systems.

The main goal in the design of Puma was scalability. MP machines with several thousand nodes need an operating system that does not grow in size or time consumption as more nodes and applications are added. For example, maintaining a buffer in kernel space for every other node in the system is not scalable. Great care has been taken to keep all buffers and data structures needed for communication in user space. An application builds the necessary structures to communicate with only the nodes it spans. Specific communication patterns inherent and known to the application can be exploited to keep the amount of memory used as buffers to a minimum.

Because of the properties of an MP environment and the demands of high-performance applications, Puma is a minimal operating system. It attempts to keep resource management overhead to a minimum while still protecting the system's integrity and shielding user applications from each other.

Many services routinely offered by other operating systems are implemented in user level libraries. This has the advantage that applications only "pay", in the form of unavailable memory and decreased performance, for the services they require.

Puma is backward compatible with its predecessor SUNMOS, the Sandia and University of New Mexico Operating System. SUNMOS has been quite successful. It currently holds the world record for the MPLINPACK benchmark performance [10], [14]. Further, the application that won the 1994 Gordon Bell award also ran under SUNMOS [36].

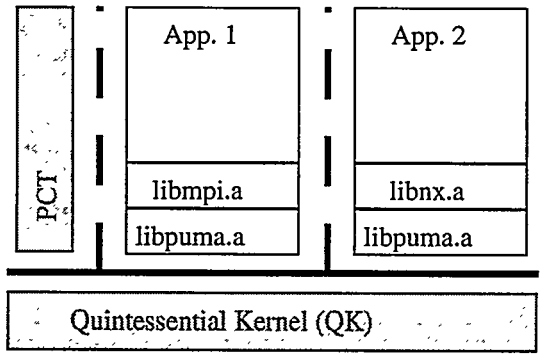

Figure 5: **Puma Node Configuration**: The Puma operating system on each node in the machine consists of the quintessential kernel (QK), a process control thread (PCT), and libraries linked with the applications running on the node. Only the QK runs in privileged mode. The QK, PCT, and each application run in separate address spaces. The solid, horizontal line separates user mode from supervisor (privileged) mode, and the dashed, vertical lines separate address spaces.

On each node, Puma consists of a quintessential kernel (QK), a process control thread (PCT), and various libraries (see Figure 5). The QK is kept as small as possible and is the only part of Puma that runs in privileged mode. Most of the functions traditionally associated with an operating system are contained in the PCT. The PCT is responsible for memory management and process control. It runs at user level, but has more privileges, in the form of QK services, than a regular application. One of Puma's main goals is to move functionality from the QK into the PCT or into the user libraries whenever possible. The PCT establishes the policies while the QK enforces them. The principle of separating mechanism and policy dates back to Hydra [38] and Per Brinch Hansen's nucleus [16].

The nodes of an MP system running Puma are grouped into service, I/O, and compute partitions (Figure 6). The nodes in the service partition run a full-featured host OS to enable users to log into the system, perform administrative tasks, and start parallel applications. Nodes which have I/O devices, such as disks attached to them, are logically in the I/O partition. They are controlled by the host OS or Puma[3]. The compute partition consists of nodes dedicated to run parallel applications. A copy of the Puma QK and PCT run on each node in the compute partition.

At the lowest level, Puma provides a send operation to transmit data to other nodes in the system, and portals to receive messages. Portals let a user level application or library define the location, size, and structure of the memory used to receive messages. Portals can specify matching criteria and the operation (read or write) applied to the user memory. We discuss portals in more detail in the next section.

Puma is currently undergoing first user testing on the Intel Paragon. Intel SSD is in the process of porting and productizing Puma for the Pentium Pro nodes of the Teraflop system being built for DOE.

7 Puma Portals

Message passing performance is an important aspect of MP machines. We have seen in Section 3 that even a single memory copy can severely impact performance. For this reason, Puma portals have been designed to allow data transfers directly from user memory on one node to user memory on another node.

While MPI is gaining acceptance, there are still many other message passing schemes in use. Furthermore, new paradigms, such as active messages and one-sided communications, are created almost daily. To get the highest performance possible, library and runtime system writers want access to the lowest level message passing mechanism available on any given machine. Puma portals must be flexible enough to support a wide variety of message passing and related mechanisms.

[3]The current implementation of Puma on the Intel Paragon uses OSF 1/AD on the I/O nodes. An earlier version of Puma on the nCUBE 2, ran Puma on the disk nodes, since a device driver was already available from SUNMOS.

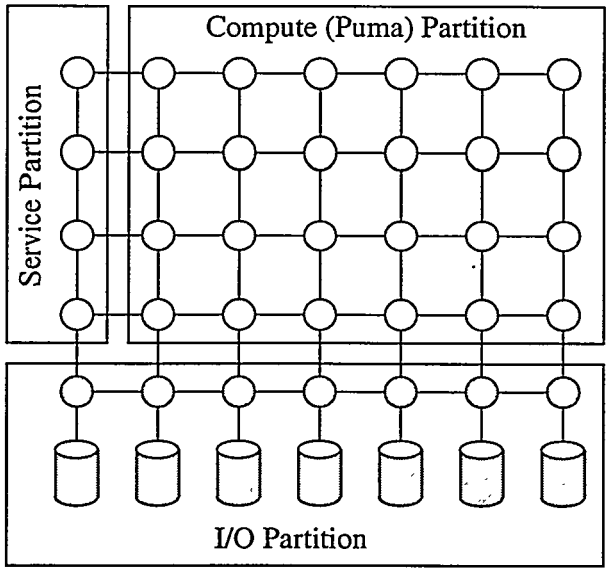

Figure 6: **System Partitioning:** A full-featured operating system is needed in the service partition to allow users to logon and to start parallel applications. In the case of the Intel Paragon, the Paragon OS, a variant of OSF 1/AD, authenticates users and provides full distributed Unix services. The Puma QK and PCT run on each node in the compute partition. The operating system running in the I/O partition could be Puma or another operating system with the appropriate driver for the I/O device and the capability to exchange messages with Puma.

During the design of Puma portals, we paid close attention to existing and proposed message passing schemes. Through case studies, we made sure that our ideas could be implemented efficiently and are sufficient to support any message passing paradigm. A Puma portal consists of a portal table, possibly a matching list, and any combination of four types of memory descriptors. We regard these pieces as basic building blocks for other message passing paradigms. A library writer or runtime system designer should be able to pick the appropriate set of pieces and build a communication subsystem tailored to the needs of the particular library or runtime system being implemented.

As a proof of concept, and to make Puma more user friendly to application programmers, we have implemented MPI, Intel NX, and nCUBE Vertex emulation libraries, as well as collective communication algorithms using Puma portals as basic building blocks. Work is currently under way to port runtime systems, such as Cilk [6] and Split-C [9], on top of portals. Work is also in progress implementing Puma portals in other operating systems such as Linux and OSF 1/AD.

Puma portals have been designed to be efficient, portable, scalable, and flexible to support the above projects. We will now look at Puma portals in more detail. We will discuss the portal table, the memory descriptors, and the matching list. We will then give a simple example of how these building blocks can be combined to present a message passing system, like Intel's NX, to an application. This section will close with a discussion of portal event handlers.

7.1 The Portal Table

A message arriving at a node contains in its header the portal number for which it is destined. The kernel uses this number as an index into the portal table. The entries in the portal table are maintained by the user (application or library code) and point to a matching list or a memory descriptor (Figure 7).

If a valid memory descriptor is present, the kernel sets up the DMA units and initiates transfer of the

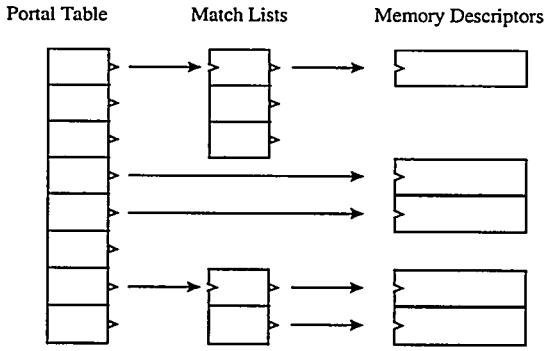

Figure 7: **Portal Table:** Messages are sent to a portal number which is an entry in the portal table. A matching list or a memory descriptor is attached to an active portal table entry.

message body into the memory descriptor. If the portal table entry points to a matching list, the kernel traverses the matching list to find an entry that matches the criteria found in the current message head. If a match is found and the memory descriptor attached to that matching list entry is valid, then the kernel starts a DMA transfer directly into the memory descriptor.

User level code sets up the data structures that make up a portal to tell the kernel how and where to receive messages. These data structures reside in user space, and no expensive kernel calls are necessary to change them. Therefore, they can be rapidly built and torn down as the communication needs of an application change.

The kernel must validate pointers and indices as it traverses these structures. This strategy makes these structures somewhat difficult to use, since the slightest error in setup forces the kernel to discard the incoming message. Most users will not use portals directly, but will benefit from their presence in libraries.

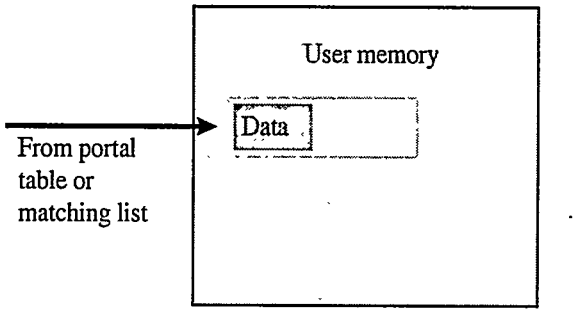

Figure 8: **Single Block Memory Descriptor**

7.2 Single Block Memory Descriptor

Four types of memory descriptors can be used by an application to tell the kernel how and where data should be deposited. This method gives applications and libraries complete control over incoming messages. A memory descriptor is laid over the exact area of user memory where the kernel should put incoming data. Most memory copies can be avoided through the appropriate use of memory descriptors.

The least complex memory descriptor is for a single, contiguous region of memory (Figure 8). Senders can specify an offset within this block. This descriptor enables protocols where a number of senders cooperate and deposit their individual data items at specific offsets in the single block memory descriptor.

For example, the individual nodes of a parallel file server can read their stripes from disk and send them to the memory descriptor set up by the user's I/O library. The library does not need to know how many nodes the parallel server consists of, and the server nodes do not need to synchronize their access to the user's memory.

Several options can be specified with a single block memory descriptor. In the parallel file server example, the offset into the memory descriptor is specified by the sender. Alternatively, the application that sets up the memory descriptor may control the offset. Instead of writing to the memory descriptor, other nodes have the option to read from it. It is also possible to have the kernel generate an acknowledgment when data is written to a portal.

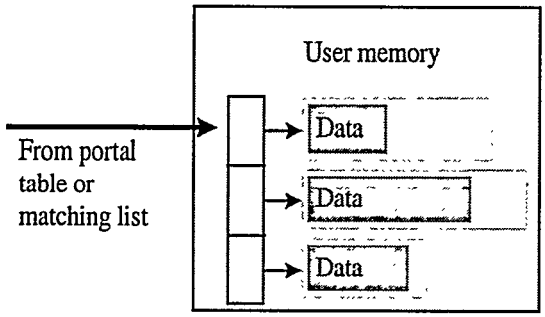

Figure 9: **Independent Block Memory Descriptor**

7.3 Independent Block Memory Descriptor

Figure 9 shows an independent block memory descriptor. It consists of a set of single blocks. Each block is written to or read from independently. That is, the first message will go into the first block, the second message into the second block, and so forth.

With a memory descriptor, if a message does not fit, it will be discarded and an error indicator on the receive side will be set. This is true for each individual block in the independent block memory descriptor.

No offset is specified for this type of memory descriptor, but it is now possible to save the message header, the message body and header, or only the message body. The user also specifies whether the independent blocks should be used in a circular or linear fashion.

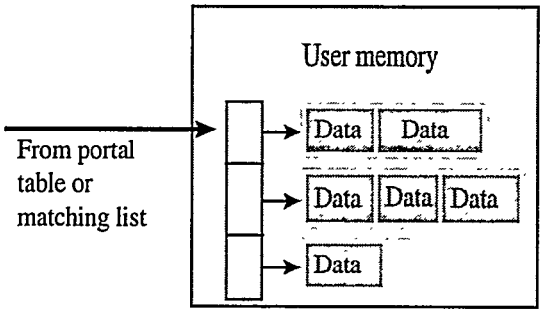

Figure 10: **Combined Block Memory Descriptor**

7.4 Combined Block Memory Descriptor

A combined block memory descriptor is almost the same as an independent block memory descriptor (Figure 10). The difference is, that data can flow from the end of one block into the next one in the list. A single message long enough to fill all blocks in a combined block memory descriptor will be scattered across all blocks. If the memory descriptor is read from, it can be used in gather operations.

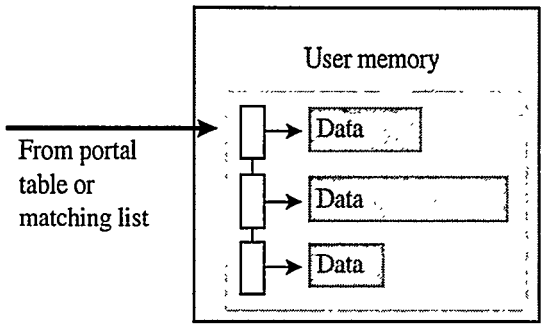

Figure 11: **Dynamic Memory Descriptor**

7.5 Dynamic Memory Descriptor

The last memory descriptor is the dynamic memory descriptor (Figure 11). Here, the user specifies a region of memory and the kernel treats it as a heap. For each incoming message, the kernel allocates enough memory out of this heap to deposit the message.

This memory descriptor is not as fast as the others, but it is very convenient to use if a user application cannot predict the exact sequence, the number, or the type of messages that will arrive. It is the user's responsibility to remove messages from the heap that are no longer needed.

7.6 Matching Lists

A matching list can be inserted in front of any memory descriptor. This list allows the kernel to screen incoming messages and put them into a memory descriptor only if a message matches the criteria specified by the user.

Matching occurs on source group identifier, source group rank, and 64 matching bits. A 64-bit mask selects the bits that must match the 64 match bits. Source group identifier and source group rank can be wild-carded.

The matching list consists of a series of entries. Each points to a memory descriptor into which the message is deposited if a match occurs. The entries are triply linked. If there is no match, the kernel follows the first link to the next match list entry to be checked. If a match occurs, but the message is too long to fit into the memory descriptor, then the kernel follows the second link. If the memory descriptor is not valid, the kernel follows the third link.

Building a matching list with the appropriate set of links and memory descriptors allows the implementation of many message passing protocols. We look at an example in the next section.

7.7 Portal Example

Figure 12 shows how the elements described in earlier sections can be combined to implement a message passing protocol.

Messages that are preposted by the user are inserted into the matching list. When a message arrives, the kernel goes through the matching list and tries to pair the message with an earlier receive request. If a match is found, the message is deposited directly into the memory specified by the user.

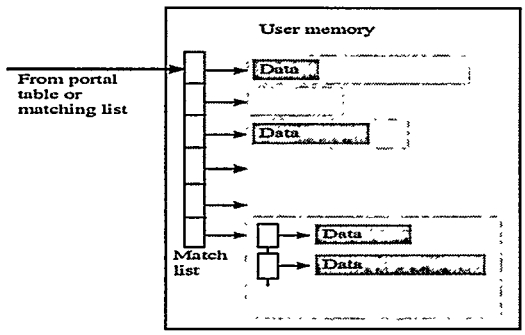

Figure 12: **Portal Example**

If the user has not posted a receive yet, the search will fail, and the kernel will reach the last entry in the matching list. It points to a dynamic memory descriptor. It is used as a large buffer for unmatched incoming messages. When the user issues a receive, this buffer is searched first (from user level). If nothing appropriate is found, the receive criteria are inserted into the matching list.

More complex and robust protocols can be built. For example, instead of storing the whole message in the dynamic memory descriptor and possibly filling it up very quickly, another scheme can be used. A second dynamic memory descriptor can be added at the end of the matching list. If the first one fills up, the kernel will continue down the matching list and then just save the message header in the second dynamic memory descriptor. When a receive is posted for one of these messages, the protocol can then request that the body of that message be sent again.

7.8 Portal Event Handlers

Each portal table entry can have an event handler. This handler is run after the data has been deposited into the memory descriptor or been discarded. The handler runs in user space, and a full context switch is necessary. This switch is very expensive on a CPU such as the i860 used in the Intel Paragon.

We are currently working on an extension to Puma that lets users install these handlers inside the kernel. Software based fault isolation or interpretation will be used to ensure the integrity of the kernel and the rest of the system.

8 Puma Design Influences

In this section we describe specific Puma features that have been influenced by the differences between distributed and MP systems.

Puma can multi-task applications on a node. However, we anticipate that high-performance applications will not make use of this feature. It is possible to load a PCT that has no multitasking support at all. The gang scheduling problem disappears in a single tasking environment. Even when Puma is run with a multitasking PCT, portals help to alleviate some of the problems introduced by multitasking. For example, an application sets up a portal to receive messages. From that point on, the kernel handles data reception and acknowledgments. The application does not need to be running when the message arrives.

While Puma provides much of the functionality provided by a standard Unix system, it is not completely Unix compatible. Some features were omitted because they do not scale to thousands of nodes. Other features are not required by high-performance MP applications. Many of the features left out for performance reasons are those dealing with direct user interactions or user management. User logins, password management, screen control, and keyboard or serial line protocols are examples of features left out of Puma.

Nearly all of the services a Puma application can request are routed by the library and the QK into the service partition. The Puma libraries and PCTs are aware of what services are available and which nodes in the service partition provide them. This strategy allows requests to be streamlined. Arguments are

marshaled up and the request is sent into the service partition. There are no provisions in the kernel or the PCT to try to find the services locally. The reason we can simplify Puma's design in this manner is that message passing is fast and the compute nodes do not have any devices attached.

Puma does not provide demand paged virtual memory. Most MP systems do not have a disk attached to each node. Therefore, paging would be prohibitively expensive and would interfere with the activities of other nodes using the same network paths and disks. Well designed applications can better determine which memory pages are not needed anymore. These pages can be filled with more data from disk. Taking advantage of high-performance I/O and network access is much more efficient than a general memory page replacement strategy implemented in the operating system [37].

Under Puma, an application can send messages to any other node in the system. The receiving kernel checks whether the destination portal exists and whether the sending process has the right to send to or receive from that portal. This improves send performance and requires no state information on the sending node. For example, there is no complicated protocol to ensure that the receiving process will accept the message or that the receiving process even exists. Performing the few checks that are necessary to ensure integrity of the system, can be done faster on the receive side because information about the sender (from the message header) and information about the receiver (from the process' control structures) is available to the kernel at the time it needs it to make the decision where to put the message or whether to discard it. Eliminating message authentication is only possible, if the network can be trusted.

The main purpose of Puma portals is to avoid memory copies. In an environment where network speed is equal or greater than the memory copy speed, this is an absolute requirement. A single memory copy at the same rate as the data streams in from the network, halves the achievable bandwidth.

Puma builds on the assumption that the nodes are homogeneous. There are no provisions in the QK to handle byte swapping or to convert to other protocols. This leads to a very shallow protocol stack and allows streamlining of message passing operations.

About one-third of the QK is devoted to handling messages, including code to deal with the portal structures and highly tuned code to access the network interface. Since the kernel is small and only a few different types of nodes are supported, the message passing code can be tuned for each architecture. Optimizations, such as preventing the CPU from accessing the memory bus while the DMA engines transfer data from the network into memory and making sure that the cache contains a messages hader that is being assembled, are possible.

A homogeneous environment also allows Puma to efficiently access unique resource, such as the second CPU on each Intel Paragon node. Under Puma, it is possible to use the second CPU as a message co-processor or as an additional compute processor. In the first case, the two CPUs exchange information through a shared memory region. One of the CPUs is always in the kernel and handles the message passing. The other CPU remains at the user level and runs the application. In the second mode, both CPUs are at the user level running individual threads of the application. One of the CPU traps into the kernel to send and receive messages on behalf of both threads [22].

For each application, Puma builds a node map that gives the application exact information about the location and distances of each node on which the application is running. We mentioned earlier that this information is very important for applications that need to optimize communication patterns. Puma can provide this information easily because the environment is static.

Puma and portals provide a very basic message passing paradigm. Everything else is built on top of that. This core functionality is available to all applications. Most of the time, a library, such as our Intel NX and MPI libraries, hide the idiosyncrasies of the Puma portal interface from the user. However, to get the very best performance, applications can access this lowest level directly, circumventing libraries that may provide functionality and overhead not desired by a particular application.

Node allocation is done by the program that loads the application. Entire nodes are allocated and assigned to the same user. The low users per node ratio and the static configuration simplifies this procedure in Puma.

Traps and interrupts on a Puma node can be highly optimized. Since the nodes are homogeneous, only one kernel version that is optimized for the given architecture is used throughout the system. Since the kernel is small and relatively simple, certain optimizations, such as not using any floating point operations in the kernel, can be implemented easily. This method removes the need to save the floating point pipelines on kernel entry and exit. On the i860 XP of the Intel Paragon, this saves at least $45\mu s$ per save or restore.

9 Related Work

Enslow [13] gives the classical definition for a distributed system. He explains in detail what features a distributed system should have. Tanenbaum [30] gives a very good introduction to distributed systems and provides an in-depth look at four systems that were being researched in the mid-80s: the Cambridge Distributed Computing System, Amoeba, V, and Eden. The Tanenbaum paper compares those four systems in such areas as naming, communication primitives, resource management, fault tolerance, and the services each system provides.

Network of workstations are distributed systems with an emphasis on higher-performance. Some of the requirements for a truly distributed system are relaxed in favor of a more efficient implementation. For example, demanding that all workstations in a NOW are of the same type and similarly equipped, not completely hiding the underlying architecture, and giving user applications more control over resources are often found in NOWs, but not distributed systems.

In [2] we find this more modern view of distributed systems and a list of issues that need to be addressed to bring NOW performance closer to the MP level. The goal is to get the performance of an MP system at lower cost.

There have been many distributed operating systems whose goal is to make a group of computers appear to a user as if it were a single system. Some of the better known ones are Eden [28, 5] and Amoeba [31], and more recently systems built on the Mach [1] kernel. A modern object-oriented operating system designed for MP computers is PEACE [26]. PEACE aims to achieve the goals of a parallel operating system as described in this report. However, PEACE also retains some features that are characteristic of a classical distributed systems.

Amoeba is somewhat of an hybrid. Its foremost goal is to provide system transparency and other key features of a distributed system. However, it also tries to be a parallel system. It allows users to pool groups of CPUs and use them for parallel applications, much as one would on an MP system.

There are few parallel operating systems. Most operating systems running on MP machines are derivatives of Unix [25, 20]. Usually a message passing system is integrated into the OS to make the fast network accessible to user programs. Often, aspects of distributed systems are also present. For example, the service partition operating system on the Intel Paragon is a version of OSF 1/AD running on Mach. It provides a single system image that gives processes on any service node the illusion of a single file system and a single, large node.

Several papers describe MP systems and their networks [18, 7, 11, 21]. The need to lower software overhead to access MP networks as well as in distributed systems with high-performance network interfaces has been noted in several papers. A thorough treatment of this topic can be found in [32]. A theoretical model that takes real-world aspects of message passing, such as latency and message transmission overhead, into consideration is described in [8].

Recent mechanisms to exploit new network technology and lower the system software overhead include active messages [34], Illinois Fast Messages [24], U-Net [33], and Puma portals [35, 27].

There is also a movement in the operating systems community to move the address space boundary between kernels and user applications to a lower level [12] or above (parts) of the application [4]. In both cases the hope is to avoid expensive context switches between kernel and user level. This should also improve message passing performance and lower overhead and message latencies.

Conclusion

There are important differences between distributed and MP systems. Most that are observable by users and applications are caused by differences in the node architecture and the network. While distributed and MP systems are converging, and some of these differentiating characteristics will disappear, there remain a few will always separate these two types of parallel systems. This separation is most notable in bisection bandwidth, network topology and transparency, configuration of the system, and location of peripheral devices.

An operating system must take these differences into account. An operating system cannot be the same on an MP and a distributed system. To get the highest performance possible on an MP system, the operating system must take advantage of unique MP features and provide applications with information that lets them optimize their communication patterns.

Dedicated NOWs have characteristics of distributed as well as MP systems. Since they are more performance oriented than a distributed system that makes otherwise idle resources available, these NOWs should be controlled by an operating system designed for an MP environment.

In the second part of this report we have introduced Puma, an operating system specifically designed for an MP environment. We have looked at some specific features of Puma and showed what differences between MP and distributed systems were responsible for particular design choices in Puma.

At the beginning of this report we quoted a definition for a distributed operating system. Another, more detailed, but agreeing definition can be found in Enslow's paper [13]. In our view, a massively parallel operating system can be characterized as follows:

> A **parallel operating system** allows a user run an application explicitly on multiple nodes. It explicitly provides resource information such as topology, available physical memory, and nodes allocated, to the application. It provides efficient message passing primitives and leaves as much of the resource control to the application as possible.

Since scalability is important, the following constraint must be applied:

> A **scalable parallel operating system** is a parallel operating system whose size and time requirements do not grow significantly when used on a larger (i.e. more nodes) system.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *USENIX Conference Proceedings*, pages 93–112, Atlanta, GA, Summer 1986. USENIX.

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

[3] B. N. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Proceedings of the 1992 USENIX Workshop on Microkernels*, 1992.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP 15 [29], pages 267–284.

[5] A. P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the tenth ACM Symposium on Operating Systems Principles*, pages 181–193, Orcas Island, Washington, Dec. 1985.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 207–216, Santa Barbara, CA, July 1995. Published as ACM SIGPLAN Notices, volume 30, number 8.

[7] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. D. Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In *Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Santa Barbara, CA, July 1995.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, San Diego, CA, May 1993.

[9] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[10] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical report CS-89-85, Computer Science Department, University of Tennessee and Mathematical Sciences Section, Oak Ridge National Laboratories, January 1995. Available from: file://netlib.att.com/netlib/benchmark/performance.ps.Z .

[11] B. Duzett and R. Buck. An overview of the nCUBE 3 supercomputer. In H. J. Siegel, editor, *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 458–464, McLean, Virginia, Oct. 1992. Purdue University.

[12] 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.

[13] P. H. Enslow. What is a "distributed" data processing system? *IEEE Computer*, 11(6):13–21, Jan. 1978.

[14] L. A. Fisk, G. Istrail, C. Jong, R. Riesen, L. Shuler, J. Bolen, A. Davis, B. Dazey, S. Gupta, G. Henry, D. Robboy, G. Schiffer, M. Stallcup, A. Taraghi, and S. Wheat. Massively parallel distributed computing. In *Proceedings of the Intel Supercomputer Users' Group. 1995 Annual North America Users' Conference*, June 1995. Available from: http://www.cs.sandia.gov/ISUG/ps/mpdc.ps .

[15] M.-P. I. Forum. MPI: A message-passing interface standard. Version 1.1, June 1995.

[16] P. B. Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–250, Apr. 1970.

[17] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of message passing and shared memory in the stanford FLASH multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 38–50, San Jose, CA, Oct. 1994. ACM Press, New York. Published as Operating Systems Review, volume 28, number 5.

[18] D. S. Henry and C. F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 111–122, Boston, MA, Sept. 1992. ACM Press, New York. Published as SIGPLAN Notices, volume 27, number 9.

[19] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 186–190, Napa, CA, Oct. 1993.

[20] Paragon XP/S product overview. Intel Corporation, 1991.

[21] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, Oct. 1985.

[22] A. B. Maccabe, R. Riesen, and D. W. van Dresser. Dynamic processor modes in Puma. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 8(2):4–12, 1996.

[23] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-Machine multicomputer:. An architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 224–35, San Diego, California, May 1993. ACM Press, New York. Published as ACM Computer Architecture News, SIGARCH, volume 21, number 2.

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

[25] J. Palmer and G. L. Steele Jr. Connection machine model CM-5 system overview. In P. U. H. J. Siegel, editor, *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 474–483, McLean, Virginia, Oct. 1992.

[26] W. Schröder-Preikschat. *The Logical Design of Operating Systems.* Prentice Hall Series in Innovative Technologies. Prentice-Hall, 1994.

[27] 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. Available from: file://www.cs.sandia.gov/pub/sunmos/papers/puma_isug95.ps.Z .

[28] *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, Pacific Grove, California, Dec. 1981. Published as ACM Operating Systems Review, SIGOPS, volume 15, number 5.

[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] A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, Dec. 1985.

[31] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–64, Dec. 1990.

[32] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[33] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In SOSP 15 [29], pages 40–53.

[34] 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.

[35] 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.

[36] D. Womble, D. Greenberg, S. Wheat, R. Benner, M. Ingber, G. Henry, and S. Gutpa. Applications of boundary element methods on the Intel Paragon. In *Proceedings of Supercomputing'94*, pages 680–684, Washington, D.C., November 1994. Paper available from: ftp://www.cs.sandia.gov/pub/papers/dewombl/Gordon_Bell_94.ps.Z .

[37] D. E. Womble, D. S. Greenberg, R. E. Riesen, and S. R. Wheat. Out of core, out of mind: Practical parallel I/O. In *Proceedings of the Scalable Libraries Conference*, pages 10–16. Mississippi State University, October 1993. Available from: file://www.cs.sandia.gov/pub/papers/dewombl/parallel_io_scl93.ps.Z .

[38] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.