Parallel Distributed Application Performance and Message Passing: A case study *

Nayeem Islam, Robert E. McGrath and Roy H. Campbell
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue,
Urbana, IL 61801

Abstract

This paper discusses experimental results concerning the design of message passing system software for shared memory and distributed memory multiprocessors as well as networks of workstations. It compares the performance of example applications and simple benchmarks running on the same operating system and message passing system on a shared memory Encore Multimax multiprocessor, on the distributed memory Intel iPSC/2 and on a network of SPARC-Station IIs connected by an 10 megabit/second Ethernet. The applications and benchmarks are compared both using the same underlying operating system and using different vendor supplied operating systems.

Our results show that CPU speed, the effectiveness of the compiler in producing efficient code and the message passing system implementation are, in order, the most important factors in application performance for the applications we examined. Based on our results we conclude that software processing within the message passing system is a major factor in parallel application performance and that this overhead can be reduced by customizing the message passing system for the application.

1 Introduction

Shared memory multiprocessors, distributed memory multicomputers, and networked uniprocessors represent three important architectures that are in use today to support tightly-coupled parallel applications. Comparisons of different computer multiprocessor and multicomputer architectures are difficult because such systems often use different message passing system and operating system software. However, such comparisons are important for the design of future systems. In this paper, we compare an Encore Multimax NS32332 shared memory machine, the Intel iPSC/2[1] hypercube, and a network of SPARCStation IIs using the same operating system, message passing system interface, applications, and benchmarks.

The research uses a portable operating system and message passing system to simplify comparison between the different architectures. The portable systems are customized to optimize resource management and allocation on the different architectures. Intel NX/2 hypercube applications and benchmarks are used in the comparison. The message passing system primitives presented in this

*This work was supported in part by NSF CDA 8722836 grant number 1-5-30035 and the first author was supported in part by an IBM Graduate Fellowship.

[1] "iPSC" is a registered trademark of Intel Corporation.

paper are based on those used in the NX/2 operating system. For control purposes, we compare our results to the NX/2 on the iPSC/2 and PVM [19] running on the SparcStation II under SunOS 4.1.3. We examine the design alternatives for message passing systems on the three different computer hardware architectures and discuss our results.

In a previous paper, we examined the issues involved in building a message passing system for a shared memory multiprocessor[12]. No one message passing system implementation proved best for all the applications used in our evaluation. In addition to application behavior, message passing system performance depended upon a number of factors including buffer copying and synchronization.

In this study, we compare the shared memory results with those for a distributed memory multicomputer and a network of uniprocessors. Once again, the performance and scalability of applications on such machines is determined by the message passing system design and the integration of message passing support into the long term scheduler of the native operating system. However, it is clear from our results that the performance of the CPU, compiler and communication hardware interact to cause large differences in overall application turn around time.

In the experiments we describe the same applications are run, without change, on all the different architectures. We perform our experiments on *Choices* an object-oriented operating system[3, 4, 5]. The different message passing systems on the common operating system are built to reuse code using object-oriented techniques in order to minimize biasing the results with the effects of different coding strategies for different systems. Class inheritance documents the reuse of such code. The systems are implemented as object-oriented class libraries which helps to structure, and thereby simplify, the coding of the different message systems. Reuse also reduces programmer overhead. Different subclass specializations of particular abstract classes document design differences. The resulting class library captures all of the design differences and similarities between the message passing systems explicitly. Further, the class hierarchies in the library form a framework for assembling new message passing systems built by reusing the components we have tested. This approach allows us to experiment with different styles of message passing quickly by reusing existing code in new designs in a systematic manner.

Comparison of Systems						
System	Processor	Clock Rate	MIPS	Cache	OS	Coding
iPSC/2	Intel 80386	16 MHZ	4	64 k	*Choices*	C++
iPSC/2	Intel 80386	16 MHZ	4	64 k	NX/2	C
Multimax	NS32332	15 MHZ	2	64 k	*Choices*	C++
SPARCStationII	SPARC	40 MHZ	28	64 k(virtual)	*Choices*	C++
SPARCStationII	SPARC	40 MHZ	28	64 k(virtual)	*VirtualChoices*	C++
SPARCStationII	SPARC	40 MHZ	28	64 k(virtual)	*PVM*/UNIX	C

Table 1: Comparison of Systems

We present performance measurements for two applications, including a ring message passing program commonly used for benchmarking message latency, and a parallel version of the Fast Fourier Transform (FFT)[15] adapted to the hypercube. Table 1 shows a comparison of the performance parameters of the hardware architectures used in our study. Table 2 compares the interconnection networks used in our study. These numbers are from manufacturers specifications.

Comparison of Interconnection Networks					
System	Network	Topology	mbits/sec	reliable	Routing
iPSC/2	Proprietary	Hypercube	> 20	Yes/Sequenced	Virtual Circuit
Multimax	Ethernet	Bus	10	No	Connectionless
SPARCStationII	Ethernet	Bus	10	No	Connectionless

Table 2: Comparison of Interconnection Networks

2 Background

Few studies compare the performance of parallel applications across architectures in order to investigate operating system dependencies. An interesting study by Lantz, Nowicki, and Theimer [14] uses distributed graphics applications to show that processor speed, amount of communication, transport protocol, communication bandwidth, and the kind of network are the dominant factors, in order of effect, on the performance of distributed applications. In particular, they conclude that with the proper design of high-level protocols network bandwidth becomes irrelevant. They consider a only limited number of platforms.

Besides UNIX, few operating systems have been ported to a large number of hardware architectures. UNIX is not suited for parallel application studies. Some parallel application studies are reported for V[6] and Orca [2]. Again, only a few platforms are considered in these studies.

Library packages such as PVM [19] allow different types of message passing applications to be run on general purpose machines running a general purpose operating system. However they have little support for process control and co-scheduling is not supported.

The NX/2 operating system includes a set of message passing primitives for asynchronous and synchronous communication[17]. In the hypercube, message transfer is reliable but messages may be received out of order. It implements typed and untyped, synchronous and asynchronous message passing. NX/2 only runs on Intel iPSC/2.

3 Platforms

In this section, we will describe the platforms for our experiments: *Choices* running on the Encore Multimax, Intel iPSC/2, and SPARCSStation II and *VirtualChoices*[3], a version of *Choices* running as a UNIX application on a SPARCStation II. The various platforms are briefly described here.

3.1 Encore Multimax

The Encore Multimax 320 is a shared-memory multiprocessor. NS32081 coprocessors provide floating point processing. Each processor accesses memory through a 64K byte cache and a 100 Mbytes/sec bus. Cache accesses do not invoke processor wait states for main memory access and do not impose any Nanobus (the processor/memory bus) traffic. The cache is thus much faster than the main memory. Maintaining locality of reference to data in the cache is an essential part of achieving high performance[9].

3.2 Intel iPSC/2 Hypercube

The Intel iPSC/2 system consists of some power of two processors (nodes) connected by proprietary network hardware which implements a hypercube topology. Each node of the cube has an Intel 80386 processor, 4 megabytes of memory, and connection hardware to use the message passing

backplane. The interconnection hardware has two major functions: routing and delivering messages. The hardware implements a form of wormhole routing that establishes a virtual circuit from the source to the destination. The message is then transferred over the virtual circuit *via* a DMA processor[1, 7, 16, 10].

The nodes of the cube run an operating system such as the Intel NX/2 [17]. Applications on the nodes of the hypercube have a simple execution environment, consisting of a processor, memory, and message passing. The node operating system provides access to the message passing hardware, multiplexes and demultiplexes messages by destination node, process, and type, and implements the "asynchronous" behavior of the message passing primitives. The node operating system also implements software protocols for variable size messages and multicasts.

3.3 Sun SPARCStation II configuration

The SPARCStation II is a 40 MHZ machine, that has a 64K virtual write back cache and 16 Megabytes of physical memory. We use a maximum of up to 4 machines for the experiments reported in this paper.

The machine is configured with the Am7990 LAN controller for IEEE-802.3/Ethernet (LANCE). The LAN controller provides for 128 send and receive buffers. It may be polled or interrupt driven. The minimum packet size is 60 bytes and the maximum packet size is 1514 bytes. The driver has facilities for multicast, which is used extensively in *Choices* for group communication.

3.4 Virtual Choices

VirtualChoices[3] is included in this paper to extend the comparison between *Choices* and PVM. Both *VirtualChoices* and PVM were run under Sun Microsystems' SunOS 4.1.3 version of the UNIX operating system. *VirtualChoices* is a port of the *Choices* operating system to the "UNIX virtual machine". It supports *Choices* applications, the *Choices* trap-based application-kernel interface, virtual memory, paging and page faults, multiple virtual Processors, disks, and interrupt based drivers for the console, timers, and networking. *VirtualChoices* was built to provide a portable, easy to use, and tool-rich prototyping environment for *Choices*. It was used to prototype the machine independent part of the message passing system.

VirtualChoices is built using two basic mechanisms: the UNIX signal mechanism is programmed to model hardware interrupts for the *Choices* kernel and the memory mapped file system is programmed to model *Choices* physical memory and the behavior of a hardware virtual memory address translation unit.

4 Message Passing System

The message passing system of *Choices* has two parts: machine dependent and machine independent. The machine independent parts of the message passing system are described in detail in earlier papers[4, 5]. In this section, we present the changes and insights we found necessary in the revision of the message passing system to accommodate networks of processors. In the next section, we outline machine dependent message passing concerns for the different platforms.

4.1 Machine Independent Message Passing

Applications send messages to named *MessageContainer*s which buffer the messages until they can be received. The machine independent layer supports the naming of *MessageContainer*s. To

eliminate repeated binding of names to *MessageContainer*s, an application process sending a message first *looks up* the name in a *DistributedNameServer*. The nameserver returns a "handle" or *ContainerRepresentative* that contains location dependent information about the corresponding *MessageContainer*. The send method of the *ContainerRepresentative* is invoked to send a message to the *MessageContainer*. In a distributed system, the *ContainerRepresentative* is an object that is local to the application process and the *MessageContainer* may be remote.

The *ContainerRepresentative* is subclassed to distinguish local or remote message containers. It may contain one or two *Address*es to locate local and remote container representatives. In practice, an *Address* identifies a *ContainerGroup* at either a local, remote, or multicast network address. The *ContainerRepresentative* forwards messages to the appropriate *ContainerGroup*s at the local and remote *Address*es. These, in turn, forward the message to the correct local *MessageContainer*. A *ContainerGroup* will forward a multicast message to multiple local *ContainerRepresentatives*, if required. In the SPARCStation II implementation, an *Address* may correspond to a broadcast "multicast" Ethernet address. On each processor, a multicast Ethernet address identifies a specific *ContainerGroup*.

The machine independent layer performs fragmentation, re-assembly, and reliable transmission. To achieve flexibility, the client (and remote-client) state is encapsulated and defined as a class with various subclasses specializing the state depending upon the interaction paradigm. For example, the client state is subclassed to support reliable RPC, multiRPC[18] and one way communication. The client state is accessible to the application through the message library. A timer can be associated with the client state and it allows applications to program recovery for lost messages, long running transactions, or server crashes.

The machine independent layer provides default network buffer management that may be specialized by on a particular hardware platform. Buffers are allocated from a pool of free buffers. The number of buffers in the pool is determined at boot time but more buffers may be created at run-time. Other optimizations such as alignment may be preset, reducing the amount of work done when a buffer is needed and is performed by a subclass that is hardware architecture specific.

4.2 Machine Dependent Message Passing

In *Choices* a set of abstract classes define the interface between the machine dependent and machine independent layers. These abstract classes are subclassed by each port.

To interface to the machine independent layer the machine dependent layer must implement the following: a subclass of *Address*, for machine identification that is specific to the hardware, a subclass for the *DistributedNameServer* class for naming, and a subclass for the *Transfer* class to manage the actual data transfer.

In addition, those operating systems ports with interconnection networks create subclasses of *NetworkBufferFreeList* to manage network buffers and *NetworkDriver* to handle network interrupts. For each of the ports, we briefly describe each of the subclasses and their functionality.

4.2.1 Encore Multimax Message Passing

The *Choices* message passing system on the Encore Multimax has been described in previous work [12]. Briefly, it uses shared memory and various queue organizations to implement efficient message passing for different communication semantics. Various specializations of the message passing mechanisms allow different forms of buffer organization, reference and value semantics, synchronization, coordination strategy, and the location of the system in user or kernel space. In

a previous studies we concluded [11, 12] that different applications perform best with different message passing systems.

4.2.2 iPSC/2 Message Passing

The iPSC/2 message passing hardware is encapsulated in a set of machine dependent classes which are used by the machine independent message passing system to send and receive hypercube messages. The abstract classes described earlier are subclassed to provide hypercube node addressing, a simple distributed name server and a transfer mechanism. The transfer schemes on the hypercube differ from those on the other platforms and are described below.

The hypercube "receive" interrupt is handled by a *Choices Exception* object, which is raised when a message arrives. The exception handler invokes the machine independent layer to buffer the message. The hardware is reset to be ready for the next message. No separate process is used to handle the interrupt.

Choices uses a single buffer size to transmit hypercube messages. In the experiments, it is fixed as a page but it could, in principle, be any fixed number of pages. The *Choices* message passing system provides fragmentation and reassembly of large messages, allowing variable-sized messages. This differs from the NX/2, which uses fixed size messages of 100 bytes for small messages and control information and a protocol that sends larger messages in a single transmission after size negotiation.

Extra copying of a message on a send is eliminated by providing a pointer to the actual DMA buffer to be used for transmission to the top level of the protocol stack. The buffer is locked while in use to prevent concurrent access. A multiple buffer scheme eliminates all but one copy on receive. When data is received, the buffer is passed up the protocol stack and another buffer is set up to receive the next incoming data.

4.2.3 SPARCStationII Message Passing

In the SPARCStationII port, *Address* class is subclassed to implement Ethernet addressing. A scheme based on broadcast uses multicast addresses for group communication. The *DistributedNameServer* sends multicast messages to find objects during bind and lookup operations. The *Transfer* mechanism interacts directly with the *EthernetDriver* to send messages. The buffer management code is designed such that the kernel buffers are mapped to kernel space and into DVMA regions. A fixed number of buffers are used. The *NetworkBufferFreeList* is subclassed to provide the necessary functionality. The LANCE chip is programmed in interrupt mode, with 32 receive buffers and 16 send buffers. Each buffer is the size of a maximum Ethernet packet, 1514 bytes. It is created in its own segment for best performance. Each buffer is a *MemoryObject* [4] that may be mapped in and out of user space. The initialization block of the LANCE chip is not cached to prevent it from being flushed at each interrupt. The buffers are 64K byte aligned (which is the size of the hardware cache) at kernel and DVMA regions to prevent flushing the cache every time the buffers are accessed. In this architecture, a blocked receiving process will incur a context switch when a message arrives for it. This platform supports both reliable, sequenced delivery of messages as well as datagram type messages. Because reliability is such an important issue for this platform, the code is more complex on this platform than on the Encore Multimax and iPSC/2.

4.2.4 *VirtualChoices* Message Passing

VirtualChoices uses the NIT (Network Interface Tap) to provide direct access to the underlying network device. This supports multicast and sending to *Choices* operating systems running on the same machine. *VirtualChoices* also supports interrupt driven I/O for the Ethernet by catching the SIGIO signal and using non-blocking UNIX "read" and "write" calls. Like the native SPARCStationII port reliability is also an important issue on *VirtualChoices* making this port as complex.

5 Compilers

The Gnu (G++) version 1.39 compiler was used for all the results reported for the ports of *Choices* in this paper. Gcc 1.39 was used to compile PVM, the associated math libraries, and the applications used with PVM. This allowed us some degree of control over the differences between the various platforms. The NX/2 applications were compiled using the C compiler that is available as part of the standard release of NX/2 on the Intel iPSC/2. Our results show that libraries and compilers can have a significant impact on the performance of both the message passing system and application performance.

6 Benchmarks

In this section, we describe the benchmarks used to evaluate the performance of our message passing system. The first benchmark is a *message latency* benchmark which we refer to as the "ring" benchmark. The message latency of a message passing system is defined as the time for one process to send a message to another process, for that other process to receive that message and send it back, and for the original process to receive it. The ring benchmark is often used to measure message performance on the Intel iPSC/2 hypercube and uses a circular connection between processes.

The second benchmark is a parallel version of the Fast Fourier Transform (FFT) [15]. The Fast Fourier Transform is a computational technique that is used extensively in branches of engineering. In the parallel version of the FFT, the application uses P processors. For a data set of n, the application has $log_2(n)$ butterfly stages, $log_2(P)$ of which are performed over the interconnection network. The $log_2(P)$ network stage involves the transfer of $16 \times n \times log_2(P)$ bytes. The FFT experiment is interesting as a scalability study. As the size n of the data increases, the message size increases linearly with n, but the computation increases as $nlog_2(n)$. Table 3 shows the sizes of messages for the different data set sizes and number of processors used in the FFT experiments.

Message Size Distribution for FFT				
Processors	FFT Points			
	128	256	512	1024
2	512	1024	2048	4096
4	256	512	1024	2048
8	128	256	512	1024

Table 3: FFT message data size distribution for varying numbers of processors

Compiler Benchmarks To evaluate the various compilers on the different architectures, we measured a few key operations that are of importance to the performance of the message passing system. The benchmark measures function calls with 0 to 5 arguments, and a variety of array manipulation operations. The inner loops of FFT were used to compare the floating point performance of the compilers.

7 Performance Experiments

We compare both the performance of message passing latency and a parallel application on the four *Choices* platforms. We then compare our results to the NX/2 running on the Intel iPSC/2 and PVM on a network of SPARCStationIIs.

7.1 Message Latency

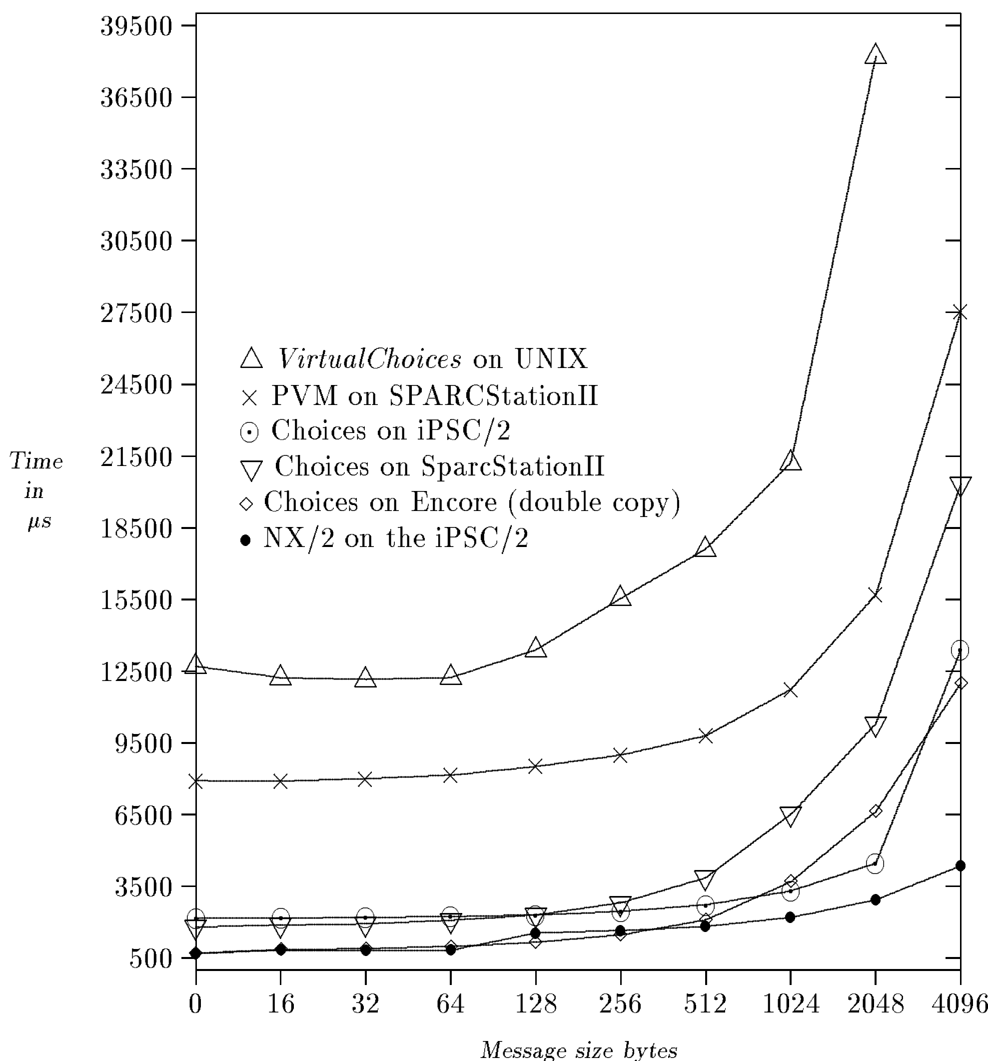

Figure 1: Comparison of Existing Systems with *Choices* IPC.

The ring benchmark provides a basic measurement of the performance of the message passing system. It measures the latency of messages of different sizes sent between two processes on two separate processors. In this section, we describe the results of the ring message passing benchmark. Figure 1 shows the message latency for *Choices* running on the iPSC/2, Encore Multimax, and SPARCStation II. The Encore Multimax version has the lowest latency, followed by the SPARC, iPSC/2, and *VirtualChoices* versions. At a message size of over 128 bytes, the SPARC version has a higher latency than the iPSC/2.

The low latency for the Encore Multimax measurements can be attributed to the multiprocessor's shared memory and the absence of network interrupts, transfer time, and reliability measures. The SPARCStationII implementation parallels the Encore Multimax until the message size requires the system to fragment the messages for transmission over the Ethernet. The SPARCStationII port fragments user messages over 1400 bytes. For large messages, the implementation uses a simple blast protocol. The *Choices* hypercube performance remains fairly flat for messages less than 4096 bytes and then it increases rapidly because of fragmentation. The hypercube implementation sends packets of variable size up to a page in length and then fragments messages into page length packets. *VirtualChoices* has the highest latency and this is caused by the UNIX scheduling and paging overhead. We note that the Gnu g++ compiler produces very different code for the various architectures.

Figure 1 compares the message latencies for PVM on UNIX and NX/2 on the iPSC/2 with the *Choices* results. For small message sizes, the Encore Multimax and Intel NX/2 message passing implementations yield similar results. As the message size increases, the NX/2 message passing system performs better. The difference can be attributed to the speed of the interconnect, the difference in processor speeds, the implementation of the highly specialized protocol of the NX/2, and the efficient code produced by the Intel C compiler. In general the curves for each of the experiments are similar in shape. This, we attribute to the message latency being dominated by data copy and fragmentation overhead.

Round trip message times for the Ring Program in μsec for Choices message Passing							
Number	Activity	Multimax	%	iPSC/2	%	SPARCStationII	%
4	Proxy calls	72	45	127	23	61	14
4	Endpoint Lookup	30	19	25	5	23	5
	Virtual functions	6 ($\times$12)	11	4 ($\times$20)	3.7	2 ($\times$28)	3
4	Network Interrupt Handler	-	0	170	31	103	23
2	Network Transmission	-	0	25	2.3	48	5
	Protocol Processing	160	25	752	35	900	50
	TOTAL	640	100	2170	100	1800	100

Table 4: Overhead for Null Message on a Variety of Architectures

The message passing implemented by PVM on UNIX gives the largest latencies of all except for *VirtualChoices*. The large latencies for PVM are explained by several internal data copies, the use of xdr for machine independence, the use of UNIX TCP sockets and UNIX scheduling. To simplify comparisons, PVM was compiled with gcc using all the same optimizations used for *Choices*. However, the underlying operating system that supports communication is SunOS 4.1.3, compiled with an optimized C compiler from Sun. We suspect that using the better compiler for SunOS boosts the performance of PVM message passing. The PVM message latency is much worse than *Choices* IPC showing the effect of protocol processing. When simple protocols are required

for parallel applications, PVM imposes a heavier overhead on applications by using more complex protocols than necessary.

Table 4 shows the breakdown of the various parts of the *Choices* message passing system for the different architectures for a null user message. The percentage of the total that each subpart contributes is given in square brackets. From the budget we can compare the basic differences between the implementations of the message passing system. Many of the basic operating system functions such as the proxy call (system trap) take almost as long on the SPARC RISC processor or are not appreciably faster than a slower CISC processor. The overhead for a virtual function call is small for all implementations. The number in parenthesis is the number of virtual function calls made in a round trip. Their number increases with the complexity of the underlying protocol. The protocol processing is most complex on the SPARCStation II since reliability is a important issue for this platform and not for the other platforms. Protocol processing remains the highest cost for all architectures except the Encore Multimax where the proxy call overhead is the largest fraction of the overall cost. The relative costs of the different overheads are the same for the iPSC/2 and the SPARCStationII. The SPARCStationII version uses an additional process for walking a packet up the protocol stack. This process is missing in the other implementations and was added primarily to experiment with different protocol processing mechanisms. On the Encore Multimax and Intel, receiving a packet does not require a context switch. A context switch takes 150μsec on the SPARCStationII. Compilers and math libraries have less of an impact on the performance of message passing latency than on applications. Our simple compiler benchmark showed showed little difference between gcc and g++ on the SPARCStationII and cc was marginally faster. The effect of the compiler is less relevant on the SPARCStation than on iPSC/2 where we found the performance of cc to be twice as fast as g++ for the operations described earlier. A summary of the performance of the compilers is given in Table 5.

Comparison of Compilers		
Compiler	SPARCStationII	iPSC/2
cc	1	1
gcc	1.03-1.12	1.5-2
g++	1.03- 1.12	1.5-2

Table 5: Comparison of Compilers

8 Application Performance

Simple benchmarks often only reveal some of the bottlenecks that might effect the performance of a system. They must be complemented with measurements of actual applications in order to obtain a good understanding of the performance of a system. In this section, we present the results of running a simple application, the FFT, on all the six platforms whose message latency we evaluated.

8.1 FFT Application

The FFT application characterizes how message passing systems behave as message sizes and numbers increase. It involves computation but is communication intensive. For comparison purposes, the performance of the FFT under NX/2 on the iPSC/2 and using PVM under UNIX on the SPARCStationII is also discussed.

Figure 2 and Figure 3 show the performance of the FFT algorithm on different ports of *Choices* with varying data set sizes for two and four processors, respectively. In both figures, the SPARCStation II ports are fastest, followed by *VirtualChoices*, PVM, NX/2, *Choices* on the iPSC/2, and *Choices* on the Multimax. It is surprising that the FFT running on *VirtualChoices* under UNIX performs better than either the Intel iPSC/2 and Encore Multimax ports. It is clear that the CPU speed is the dominant factor for performance, and other experiments show that it becomes increasingly more important as the data set size increases. Since the computation varies as $nlog_2(n)$ and communication as a linear function of n, improved CPU speed has a bigger impact on the computation of the application than on message passing. The PVM and NX/2 experiments also confirm this observation. Although PVM uses TCP/IP and is less efficient than the NX/2 message passing approach, the performance of the PVM experiment is better than that of the NX/2 experiment.

Message latency does not predict well the performance of the different experiments. Message latency would suggest either the Encore Multimax or NX/2 should be faster but clearly this is not the case. For a specific architecture and hardware configuration, the latency of the message passing system does have an effect. For example, the SPARCStation II port of *Choices* performs twice as well as PVM. The results of the runs on the SPARCStationII port are also less variable, due in large part to the scheduling facilities added to aid in running of parallel programs. *Choices* locates idle workstations and schedules gangs on distributed systems.

Tables 6 and 7 show the absolute times for our experiments. The relative performance of *Choices* on the SPARCStation increases faster, as the data set size increases, than PVM and *VirtualChoices*. The port to the Encore Multimax and the iPSC/2 suffers from poor performance because, with larger data set sizes, the computation increases faster than the messages sizes. The systems with faster CPU's perform proportionally better.

Absolute times (milliseconds) of FFT with varying input data set size				
System Configuration	FFT Points			
	128	256	512	1024
Choices on SPARCStationII	22.00	36.00	56.00	116.00
PVM	27.00	39.00	78.00	174.00
Choices on Encore Multimax	300.00	480.00	1061.00	2485.00
Choices on iPSC/2	148.00	328.00	738.00	1655.00
NX/2	40.00	90.00	227.00	490.00
VirtualChoices	25.00	39.00	110.00	195.00

Table 6: Performance of FFT for 2 Nodes on Different Platforms with Varying Data Set Sizes

Table 8 shows the time spent communicating, waiting and computing on each of the six platforms for data set sizes of 128 and 1024 for 4 nodes. These times are obtained from one node only. The communication costs are derived from Figure 1. The message passing classes and libraries were instrumented to calculate the total time in the message passing system. The waiting time is the time spent in the message passing system minus the costs of communication as derived from Figure 1. The waiting time cost is the time spent in blocking receives. For small data set sizes the waiting times are small. For large data set sizes the waiting time increases. Although, each processor executes identical application code, the data values differ, and the differing times to evaluate iteratively the trigonometric functions create a communications asynchrony. This application asynchrony results in some unavoidable message passing overhead, in the form of increased wait times on blocking receives, on all systems. This overhead, as a percentage of the total time, is greatest for platforms with the fastest CPUs. Because PVM and *VirtualChoices* run under UNIX

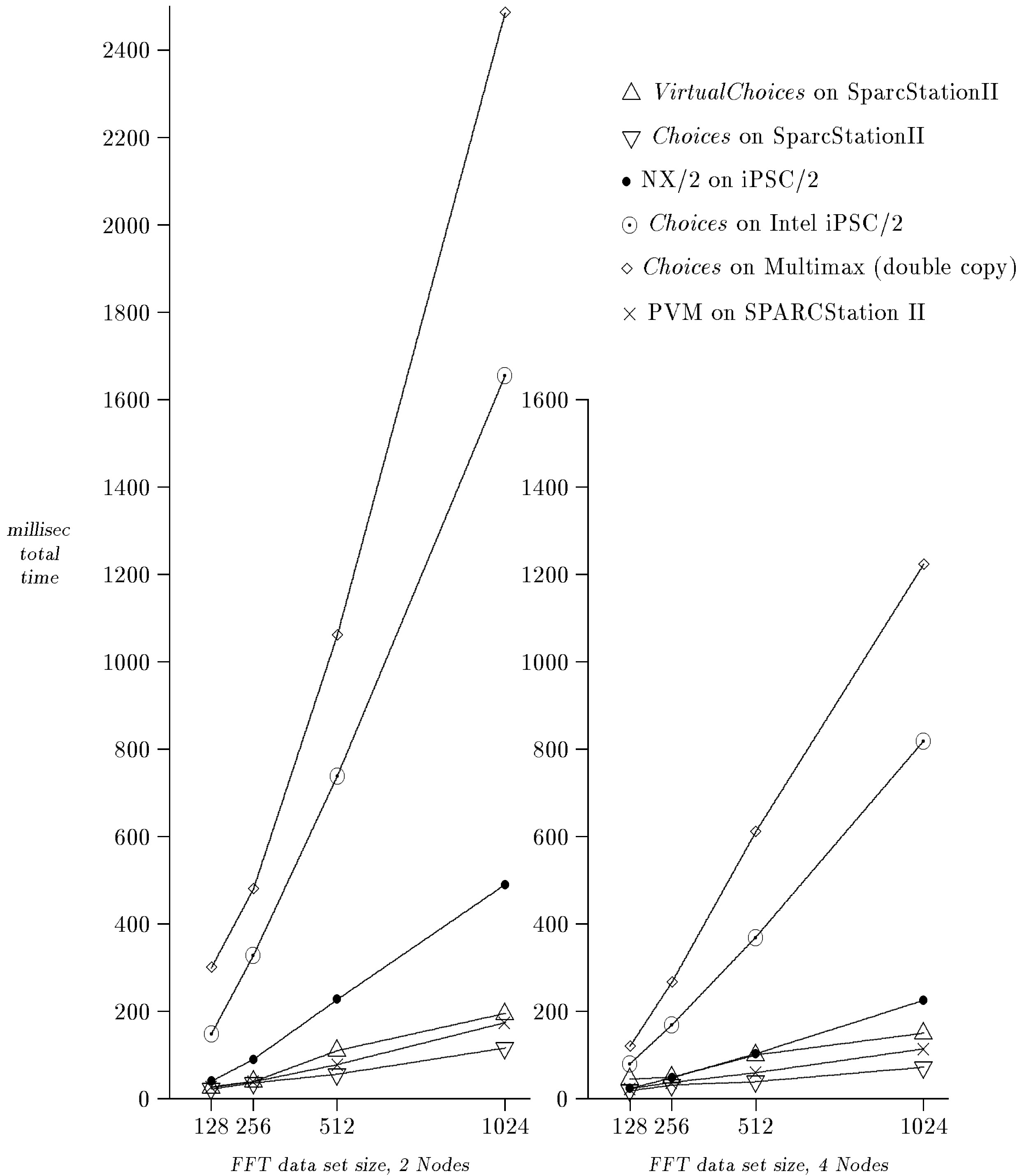

Figure 2: Execution Times for FFT 2 nodes With Varying Data Set Sizes

Figure 3: Execution Times for FFT 4 nodes With Varying Data Set Sizes

Absolute times (milliseconds) of FFT with varying input data set size				
System Configuration	FFT Points			
	128	256	512	1024
Choices on SPARCStationII	18	31.00	38.00	72.01
PVM	27.00	37.00	59.00	114.00
Choices on Encore Multimax	120.00	266.00	611.00	1223.00
Choices on iPSC/2	80.00	169.00	370.00	819.00
NX/2	23.00	48.00	103.00	225.00
VirtualChoices	45.00	49.00	100.00	150.00

Table 7: Performance of FFT for 4 Nodes on Different Platforms with Varying Data Set Sizes

System	128			1024		
	Wait	Comm	Comp	Wait	Comm	Comp
Choices on SPARCStationII	6.9	4.6	6.5	8.6	13	50.4
PVM	2.2	16.0	7.8	26.0	24.0	64.0
Choices on Encore Multimax	29.0	3.0	87.6	60.0	7.2	1155.7
Choices on iPSC/2	5.8	4.6	69.6	22.0	6.6	790.3
NX/2 on iPSC/2	3.2	3.0	16.7	5.7	4.4	214.9
VirtualChoices	12.4	26.6	8.0	38.0	42.0	80.0

Table 8: Variation Communication, Waiting and Computation Times with Increasing Data Size for FFT on 4 Nodes

and are dispatched by its scheduler, these platforms show greater waiting times for larger data set sizes.

For machines with faster CPUs, the time spent in the message passing system is proportionately larger. The CPU decreases computation time but has less impact on improving message latency. The difference in the computation times between the FFT on the iPSC/2 and other various architectures is due to the faster compilation of floating point operations by the native NX/2 compiler.

For a specific architecture, reducing message latency is important. For example, the *Choices* optimization that exploits blast protocols for large data message sizes helps to improve the performance of the *Choices* implementation as compared to the PVM implementation on the SPARCStation II.

FFT Speedup for 512 points			
System Configuration	Number of Processors		
	1	2	4
Choices on SPARCStationII	1	1.42	2.2
PVM	1	1.2	1.58
Choices on iPSC/2	1	1.81	3.56
Choices on Encore Multimax	1	1.89	3.3
NX/2	1	1.84	3.77
VirtualChoices	1	1.11	1.22

Table 9: FFT Speedup on Various Platforms

In general for the FFT, as the number of processors is increased the message sizes become smaller. As there are more messages, the performance of the message passing system becomes more important. The platforms with the better message passing systems will perform better. Again *Choices* on the SPARCStationII outperforms PVM. Both ports to the iPSC/2 exhibit remarkably similar behavior.

Figure 9 shows how the applications scale with increasing numbers of processors. Those systems with the fastest CPU scaled worst of all, since most of the time is spent in the message passing system. Within the same CPU class, the faster the message passing the better the application scales. The port of the NX/2 scales the best of all. These results confirm results of others that the lower the communication overhead as compared to the computation the better an application will scale [8].

Effect of Compilers and Math Libraries The application programs for NX/2 were compiled using the C compiler that is native on the NX/2 software and PVM was compiled with gcc version 1.39. The PVM applications were compiled with the *Choices* math libraries, which are the Berkeley public domain math libraries. Exactly the same compiler optimizations were used for PVM and *Choices*.

We found that changing the compiler and library for PVM to the SunOS 4.1.3 compiler and math library increased application performance for the 4 Node, 1024 point data set size experiment by a factor of 1.60, making it comparable to the *Choices* native port.

On the Intel iPSC/2 we found that the inner loop of the FFT ran about 5 times faster on the NX/2 using native NX/2 compiler and libraries, than when using the g++ and the *Choices* math libraries. This accounts for the difference in the computation times between NX/2 and *Choices* on iPSC/2 for the FFT application, as can be seen in Figure 8.

9 Variability

The results reported in this paper for the roundtrip ring measures are the average of a few thousand runs. The results for the FFT were run several times and averaged. However, the results for *VirtualChoices* running FFT showed a variation by a factor of about 2-3 times. Under moderate to heavy loads, PVM exhibited the same variations. For FFT under *VirtualChoices* and PVM we report only the best results. The other platforms reduce variability of the execution time of applications by providing some form of dedicated and simultaneous scheduling for groups of processes such a gang or co-scheduling [13].

10 Conclusion

In this paper, we show that several factors affect the performance of parallel message passing applications in our experiments. In order of impact, the speed of the CPU, the compiler, and the message passing system have the largest impact on performance. Within a platform the design of the message passing system can have a tremendous impact on the performance of the application. We have also described the design of a portable message passing system, discussing the overheads of the basic implementation on three hardware platforms for providing the same services to an application. This allows us to meaningfully compare a variety of message passing applications. Although we did not discuss results for scheduling in detail, gang scheduling also contributed considerably to application performance on shared memory machines as well as on distributed

systems. When using PVM in the experiments, gang scheduling is not used to organize the parallel execution of the application processes. An application may experience execution times that vary as much as a factor of 3. *Choices* has facilities for locating idle or lightly loaded groups of machines and for managing these machines[13].

References

[1] Ramune Arlauskas. iPSC(R)/2 System: A Second Generation Hypercube. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, pages 38–42, January 1988.

[2] Henri Bal, Frans Kaashoek, and Andrew Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, March 1992.

[3] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Experiences Designing and Implementing Choices: an Object-Oriented System in C++. In *Communications of the ACM*, September 1993.

[4] Roy H. Campbell and Nayeem Islam. *Choices*: A Parallel Object-Oriented Operating System. In Gul Agha, Akinori Yonezawa, and Peter Wegner, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[5] Roy H. Campbell, Nayeem Islam, and Peter Madany. *Choices*, Frameworks and Refinement. *Computing Systems*, 5(3):217–257, 1992.

[6] David Cheriton. The V Distributed System. *Communications of the ACM*, pages 314–334, 1988.

[7] Paul Close. The iPSC(R)/2 System Node Hypercube. In *Proceedings 3rd International conference on Hypercube concu rrent computers and applications*, pages 43–50, January 1988.

[8] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[9] Mark Hill and James Larus. Cache Considerations for Multiprocessor Programmers. *Communications of the ACM*, pages 97–102, 1990.

[10] Intel. *Intel 82258 Advanced Direct Memory Access Coprocessor (ADMA) [Data sheet]*. Intel, 1987.

[11] Nayeem Islam and Roy Campbell. Design Considerations for Shared Memory Multiprocessor Message Systems. Technical Report UIUCDCS-R-91-1764, University of Illinois Urbana-Champaign, December 1991.

[12] Nayeem Islam and Roy H. Campbell. "Design Considerations for Shared Memory Multiprocessor Message Systems". In *IEEE Transactions on Parallel and Distributed Systems*, pages 702–711, November 1992.

[13] Nayeem Islam and Roy H. Campbell. "Uniform Co-Scheduling Using Object-Oriented Design Techniques" . In *International Conference on Decentralized and Distributed Systems*, Palma de Mallorca, Spain, September 1993.

[14] Keith Lantz, William Nowicki, and Marvin Theimer. " An Empirical Study of Distributed Application Performance". In *IEEE Transactions on Software Engineering*, pages 1162–1174, October 1985.

[15] H. Miyata, T. Isonishi, and A. Iwase. Fast Fourier Transformation using Cellular Array Processor. In *Parallel Processing Symposium JSPP 1989*, pages 297–304, February 1989.

[16] Steven F. Nugent. The iPSC(R)/2 System Direct Connect(TM) Communications Technology. In *Proceedings 3rd International conference on Hypercube concu rrent computers and applications*, pages 51–60, January 1988.

[17] Paul Pierce. The NX/2 Operating System. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, pages 384–390, January 1988.

[18] Mahadev Satyanaryanan and Ellen Siegel. "Parallel Communication in a Large Distributed Environment". In *IEEE Transactions on Software Engineering*, pages 328–348, March 1990.

[19] V.S. Sunderam. PVM A framework for parallel distributed computing. *Concurrency Practice and Experience*, 2(4):315–339, December 1990.