Design Considerations for Shared Memory Multiprocessor Message Systems

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

Abstract

Message passing and shared variables are the two major approaches to interprocess communication. Many distributed parallel programs use algorithms based on message passing because the technique can be implemented efficiently both on distributed and shared memory multiprocessors. The performance of such parallel programs depends on the design of the message passing system.

In this paper, we study experimentally the comparative performance of different message passing system designs on a shared memory Encore Multimax multiprocessor. The systems are measured both by benchmarks and by running example distributed applications. The benchmarks and applications are Intel hypercube programs recompiled for the shared memory machine, but otherwise unchanged. To act as a control, the shared memory machine results are compared with the performance of the benchmarks and applications on the iPSC/2 hypercube. All systems are built using object-oriented techniques in order to minimize the effect of coding different message based systems. Common design decisions are inherited using subclassing. Differences in design are coded as different specializations of an abstract class, allowing easy comparison of the implementations. The design alternatives considered are buffering, buffer organization, reference and value semantics, synchronization, coordination strategy, and the location of the system in user or kernel space. The results include measurements of the effects of the design alternatives, memory caching, message sizes, copying, and gang scheduling. Our conclusions are a set of recommendations for improving the performance of message passing systems on a shared memory multiprocessor.

1 Introduction

In this paper, we examine design alternatives for message passing systems on shared-memory multiprocessors. We identify a set of orthogonal parameters that can be used to describe a message passing system. Central to the design is identifying the appropriate operating system support for each design of a message passing system. We use C++ and object-oriented design techniques to expedite the development of the components of the system. Object-oriented techniques also facilitate describing the relationships between various components.

The performance of a parallel application depends upon many factors including the algorithm, the parallel programming model, the target parallel computer architecture, the compiler, and the target operating system. Investigation of the impact of operating system design on parallel application performance is difficult because of the work required in order to make experimental comparisons. Extending an existing operating system to support various parallel programming models using a variety of different implementations without compromising efficiency is a complex and time consuming task. Alternatively, porting different operating systems to the same hardware platform in order to make comparisons is also expensive. Either approach may produce misleading results. Our research is concerned with building customized operating systems using

object-oriented techniques to support high-performance parallel applications. By customizing an operating system and support libraries, we compare the performance of applications for different operating system designs systematically and rapidly.

Message passing and shared variables are the two major ways of interprocess communication on shared-memory or distributed memory multiprocessors. These schemes have much in common. Parallel algorithms based on message passing and on shared variables have implementations on both shared-memory and distributed memory multiprocessors. Some researchers claim improved performance for algorithms if they are written using message based techniques and run on shared-memory machines[24]. There are a variety of ways to implement a message passing scheme on a shared-memory multiprocessor and it is not easy to predict the performance of any one given approach given a description of the machine architecture. In order to understand the issues that effect performance in a message passing system, we compare a spectrum of different message based systems. These systems are compared with benchmarks and applications used to measure and compare distributed memory multiprocessors known as hypercubes. In this respect, our work is different from recent schemes that are mostly aimed at improving multiprogrammed thread performance [2]. We are interested in operating system support for shared-memory machines that allows for application performance similar to that on dedicated distributed memory machines such as the Intel iPSC/2 hypercube [12]. Without dedicated resources, applications do not attain the performance of distributed memory machines. Our focus is on effectively running scientific applications in a general purpose operating system.

The designs are carefully encoded using a structuring technique known as object-oriented programming. This approach is used to record similarities and differences between their implementations. The performance of the benchmarks and applications using the message systems are compared against each other and with their performance on the iPSC/2 hypercube, which is used as a control.

We simplify the coding of the different message systems by implementing them on *Choices*, an object-oriented operating system. *Choices* is a symmetric multiprocessor operating system written almost entirely in C++. We use object-oriented programming in C++ to track the reuse of code implementing similar design components in different message passing systems. Class inheritance documents the reuse of such code and 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 messaging 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.

We present performance measurements for running several applications including a ring message passing program commonly used for benchmarking message latency, a Fast Fourier Transform (FFT)[27] adapted to the hypercube, and a Simplex program that has been analyzed in detail[36]. Six different message systems are compared using the message latency benchmark. We further investigate the effects of message data transfer on the scalability of the FFT and Simplex applications. The behaviour of the application using the various message passing implementations can be understood in terms of copying and the hardware data and instruction cache. The shared-memory machine used in the experiments is an Encore Multimax [13] with NS32332 processors. To act as a control for our experiments, the performance of the applications is compared with their performance on Intel's NX/2 operating system [30] running on the Intel iPSC/2. The results show that the relative performance of the applications on the shared-memory machine is not degraded by either the machine, *Choices*, or the object-oriented techniques used.

Our results show:

- Parallel applications are likely to exhibit a bimodal message size distribution.
- In tightly coupled parallel applications with cooperating processes it is more efficient to use spin-locks rather than semaphores, implemented by blocking, for synchronization.
- It is important for application processes to be gang scheduled as well as non-preemptable while they are running.

- It is important for the applications not to be preempted by interrupts while running. Artificial benchmarks often do not correctly predict the behaviour of applications subject to context switching overhead.
- Read, write, and no access to message data has a significant effect on the performance on message latency benchmarks.
- There are a variety of ways of porting a distributed application to a shared-memory machine. Using a message system based on pointer passing provides the best benchmark performance. However, it may require changes to the application. Further, applications may run slower on a pointer implementation than a buffer copying implementation for four or more processors.
- Whether the message system is implemented in the kernel of the operating system or in shared-memory user space is an overhead that varies with the number of messages sent per processor and remains constant as the message sizes and number of processors increase.
- The distributions of round trip time versus message size for the better message passing systems on the Encore Multimax is similar to those for the Intel hypercube for the message sizes used in the experiments.
- The memory caching scheme used in a processor has a major impact on the performance of a message passing scheme.
- Finally, our work shows that object-oriented design techniques are very useful for rapid prototyping and experimentation of different designs without sacrificing the performance of the system. We ran both the FFT and Simplex applications without modifications under *Choices* on the Encore Multimax and under NX/2 on the Intel Hypercube iPSC/2. We obtained similar behavior and speedups under both systems.

Our work differs from existing work in that we use object-oriented design techniques to compare systematically a number of different design alternatives in the implementation of an existing message system protocol. We believe our approach is more comprehensive and extensive than previous work by Bershad [5, 4] and identifies the pitfalls involved in designing a message passing system on a shared-memory multiprocessor. Our message passing protocol is based on the Intel NX/2 hypercube message passing system. The implementation uses recent results from scheduling [6] and spin-locking [1, 17] algorithm studies. We perform a set of control experiments that show that applications running on our message system have similar behavior as on the hypercube.

Despite the extensive use of C++ virtual functions in the experiments, the evidence indicates that an object-oriented operating system can be customized to achieve comparable performance to a proprietary operating system supported with purpose-built hardware. However, the results cannot be interpreted as a comparison between hardware architectures, as indicating the advantages of using a message passing paradigm on a shared-memory machine, as demonstrating that object-oriented code is more efficient than assembler, or as concluding that optimizations and customizations can be always performed using object-oriented techniques. Instead, they indicate that the object-oriented approach can be used as an experimental methodology to evaluate systematically different system design alternatives without any significant loss in performance.

The remainder of this paper discusses our platform for experimentation, *Choices* [8], the parallel processing features that were added to *Choices* to support hypercube applications, the applications and their measurements.

2 Background

In this section, we overview some of the current issues in interprocess communication and object-oriented system design. We review the message passing system protocol, operating systems, and machines that are

used in the experiments.

2.1 Parallel Message-Based Applications

Parallel applications running on shared-memory multiprocessors like those built by Alliant, Encore and Sequent and message-based parallel applications running on distributed memory multiprocessors like the Intel iPSC2 hypercube[32, 7] have demonstrated notable performance improvements over their sequential counterparts. How well interprocess communication and processor utilization scale as the number of processors available increases is a research issue[39]. The various parallel programming techniques used to program applications for different computer architectures using dissimilar operating systems make comparisons difficult.

There has been little previous research examining the performance of parallel applications running on multiprocessor object-oriented operating systems. In fact, few operating systems have been coded extensively in an object-oriented programming language. Operating systems with messaging primitives on distributed memory systems include V System[10], Amoeba[37], and NX/2[30]. Operating systems with shared-memory primitives on shared-memory machines include Umax[13] and Mach[31]. Demos[3] and Mach[31] are examples of systems that implement messaging primitives on top of a shared-memory multiprocessor. Shared distributed virtual memory[23] allows a shared-memory application to run on a distributed memory computer. Various systems like Amoeba[37], Clouds[14], Chorus and SOS[35] are object based. Object-oriented languages have been used with some success to implement parallel processing thread packages such as Amber[9].

Studies of message-based applications by LeBlanc [22] and Lin [24], on shared-memory machines reveal interesting problems and issues. LeBlanc uses different implementations of the same application and concludes that the message-based applications scale better for a small number of processors (less than 64) but for a large number of processors the shared-memory applications scale better. Snyder and Lin also rewrite applications to compare shared-memory and message passing programming models. They report experiments in which contention for memory in a shared-memory multiprocessor appears reduced if the programming model is message-based leading to better performance. They reason that this is caused by locality and large computational granularity of such programs.

The performance of a parallel application is sensitive to the scheduling mechanisms and policies of the operating system [6, 39, 29, 20]. NX/2 uses gang scheduling in which a process is dedicated to a processor for the duration of the application[30]. Unlike NX/2, the Medusa operating system implements coscheduling [29], a technique that *tries* to schedule the processes of an application to run at the same time. The Alliant scheduler[20] supports a fixed number of scheduler classes and uses a scheduling vector for each processor to indicate which classes should be searched for work in what order. Groups of processes are assigned to a scheduler class and each group within a class is run simultaneously. Blocking a particular process blocks the group of processes. Process control and scheduling issues on the Encore Multimax[39] indicate that applications can be written to negotiate with a server for an allocation of a number of processors. The server and the application together determine the level of application concurrency. In a distributed system, techniques have been proposed to locate idle processors efficiently[38, 10].

The Mach group [31, 6] identify several important parameters for gang scheduling: creation of a processor set, assignment of a processor, and assignment of a thread or a process. We present *Choices* performance numbers for these parameters.

Bershad [5, 4] has shown that fast interaddress space communication is possible on shared-memory machines. Such schemes can be implemented in the kernel or in shared-memory user space. Our work is similar in that we have implemented the NX/2 messaging primitives both in shared-memory user space and in the kernel. However, we use kernel-based context switching, object-oriented techniques for the design of the facilities and emphasize reuse. Bershad's work concentrates on shared memory machines, multiprogramming, and a simple set of message primitives. The work has not yet been applied to complex message passing systems such as found in the NX/2. It is not clear how a typed, asynchronous message primitive would be encoded efficiently using his techniques. Last, Bershad does not report measurements of any parallel hypercube-like applications using his message system. Our goal was to build a high performance message

system that could run existing code. The NX/2 primitives are very general and our experiments involve substantial applications that use most of the primitives.

Our work differs from previous work in that:

1. We identify a larger set of orthogonal parameters affecting the design of a message passing system.
2. We evaluate our work using primitive benchmarks, as well as real applications.
3. We use object-oriented design techniques.
4. We experiment on an object-oriented operating system.
5. We compare our system with a distributed memory machine that is designed for message passing.

2.2 The NX/2 Message Protocol

The NX/2 includes a set of messaging passing primitives for asynchronous and synchronous communication[30]. In the hypercube, message transfer is reliable but messages may be received out of order. In addition, the hypercube message is buffered and has flow control.

Two calls implement synchronous message passing:

1. *csend(type,buffer,length,node,pid)* and
2. *crecv(typesel,buffer,length).*

The *csend* call does not return until the message has been put on the network and the buffer specified in the message by the application is free to be used again. *csend* does not necessarily wait for a corresponding *crecv* to occur at the destination process. The *crecv* call selects an incoming message by type and receives it into a buffer. This call is blocking, it does not return until the message is in its buffer.

The corresponding asynchronous calls are:

1. *messageId = isend(type,buffer,length,node,pid)* and
2. *messageId = irecv(typesel,buffer,length).*

The *isend* call starts transmission and then returns but the buffer cannot be reused immediately. A *messageId* is returned that can be used to query the message passing system periodically to determine if the buffer is free to be reused by the application that invoked the *isend* call. There is no way to query the system to determine whether or not the message was actually received. The *irecv* call registers a request with the message passing system to receive a message into the buffer specified. The call returns immediately with a *messageId*. This *messageId* can be used to check if a message has been put into the buffer. The *irecv* call can be used to reduce message handling overhead since it informs the message passing system where to put the message when it arrives.

2.3 The NX/2 Operating System

The NX/2 operating system allows an iPSC/2 hypercube to be partitioned into smaller hypercubes dedicated to particular applications. The processes of each application are gang scheduled. The NX/2 operating system has two different protocols for short and long messages: for message sizes less than 100 bytes, the transfer uses pre-allocated message buffers. For larger message sizes, message buffers are allocated after the sender node sends a control message to the receiver. The message is sent when the control message is acknowledged and, on receipt, it can be copied directly into the application's receive buffer.

2.4 The iPSC/2 Hardware

The Intel iPSC/2 used as a control in our experiments has circuit switching hardware to implement message passing between network nodes[12]. Similar message transfer latencies and overhead are incurred no matter which two nodes of the hypercube are involved. To understand the applications, we ran them on an instrumented version of the NX/2 kernel [26]. This instrumentation did not impact performance. It measures idle time, time spent in context switching and time spent by applications in the message passing system.

2.5 Choices

Choices is a multiprocessor operating system designed as an object-oriented system using object-oriented coding techniques in the C++ language [8]. The many advantages of an object-oriented approach are reported in [25, 21]. *Choices* has, as its kernel, a dynamic collection of objects. System resources, mechanisms, and policies are represented as instances of classes that belong to a class hierarchy. The system has over 300 classes and 78,000 lines of source code. *Choices* runs on bare hardware: the Encore Multimax, the Apple Macintosh IIx, the SUN Sparcstation 2, the AT&T WGS-386, and the IBM PS/2. It supports both uniprocessor and multiprocessor architectures.

Our goal in the design of *Choices* is to allow the system to be specialized to support parallel processing primitives, different machine architectures, and different application programming models. By using inheritance, encapsulation, delegation, frameworks and other object-oriented techniques, *Choices* is intended as an easily modifiable testbed that can give comparable results for diverse applications running on a variety of computer architectures[33, 8]. All entities in the operating system are modeled as objects and include: classes, system processes, user processes, regions of memory, files, and hardware devices like CPU's and disk controllers.

Fundamental to the design of *Choices* is the notion of frameworks. Each subsystem, for example the virtual memory subsystem, is designed within a framework. A *framework* lies at the heart of any complex object-oriented implementation[41]. It consists of a set of class hierarchies and a design for how the instances of the classes from these hierarchies can be combined to make systems. The key ingredient in a framework is the identification of the abstract concepts of the problem domain rather than an actual implementation[15]. By using the framework with different implementations, it is possible to gain confidence that the abstract concepts model the problem domain closely.

In designing a subsystem such as virtual memory, a framework is proposed, an example implementation is constructed within that framework, and the framework is modified by the practical implementation. New example implementations are created and the framework is updated as necessary. The frameworks evolve by prototyping until they adequately describe the abstract properties of the subsystems that are being built.

In our experiments, we added a message subsystem to *Choices* based on the the NX/2 messaging protocols. The subsystem is implemented within a message system framework. The object-oriented design of this framework is presented in [19]. In this paper, we concentrate on the properties of the message system and how they effect the performance of the message passing system. The framework supports various message system implementations. In addition, we augmented the *Choices* process management, scheduling and application interface frameworks in order to support efficient message based parallel applications.

2.6 Encore Multimax

The Encore Multimax 320 is a shared-memory multiprocessor. A NS32081 coprocessor provides floating point processing. Each processor accesses memory through a 64k byte cache and a 100 Mbytes/sec bus. Cache accesses do not invoke the processor wait states for main memory access and do not impose any Nanobus 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[18].

The cache has 16 byte cache lines, each line is divided into 4 byte sublines. A read from the cache fetches 8 bytes from memory into the cache line. For two consecutive reads, the second read will access cached data. A write to the cache stores 4 bytes back into memory in the write-through. For two consecutive writes,

the second write performs like the first write. An initial write to a cache line may read 8 bytes first. The cache is directly mapped into memory. The lowest 4 bits of an address select the subline. The next set of bits select the line. Thus each cache is mapped modulo 64k into memory. The cache RAM has an access time of 25 nanoseconds. It takes 320 nanoseconds to fetch 8 bytes to or from memory to accommodate an uncached read; if consecutive reads are to different memory banks then the 8-way interleaved memory allows the transfer of 8 bytes every 80 nanoseconds in the absence of bus contention. An uncached write takes 120 nanoseconds because it first reads 8 bytes of data and then changes 4 bytes of data. It takes $2^{12} \times 80$ or 0.328 milliseconds to fill the cache. It takes $2^{12} \times 120$ or 0.492 milliseconds to write the entire cache. Instruction fetches can interfere with the data held in the cache because instructions are fetched into the cache.

2.7 Summary of Equipment used in Experiments

Our experiments were conducted on a 16 processor Encore Multimax with 32 megabytes of memory and a 16 node Intel iPSC/2 hypercube with 4 megabytes of memory per node. The details of the processors for each system are shown in Figure 1.

Comparison of Processors Based on Manufacturers Specifications					
System	Processor	Clock Rate	MIPS	OS	Messaging Primitives
iPSC/2	Intel 80386	16 MHZ	4	NX/2	Written largely in assembler
Multimax	NS32332	15 MHZ	2	*Choices*	C++

Figure 1: Comparison of processors

3 Design Considerations for Message Systems

The hypercube message system primitives were added to *Choices* as a specialization of a framework for message systems. The modifications include message system extensions for the NX/2 messaging primitives, gang scheduling, libraries and several small utility programs. Several changes were required to other subsystems in order to port hypercube applications to *Choices* without source code modification. The application interface on *Choices* differs from that of the NX/2, requiring the construction of a compatibility library. The NX/2 messaging primitives are defined in *Choices* using a set of abstract classes that provide a "standard" interface. Specific message system implementations are written as concrete subclasses of these abstract classes. This simplified configuring *Choices* for each different message implementation in order to measure its performance with the benchmarks. This section describes some of the many design alternatives for implementing the NX/2 message primitives on a shared memory machine. It also describes the changes to the designs of various *Choices* subsystems that were required to accommodate efficient parallel applications.

3.1 A Framework for a Messaging System

The message system framework accommodates different message passing semantics, buffering, and implementation schemes for the NX/2 message primitive interface. Six factors dictate the construction of a message system and they are represented as separate classes within the framework. These factors are:

- *Location:*
  - *User space*: the message system is implemented in user level. A send or a receive does not involve passing data through the kernel but is implemented using user shared memory. There is minimal checking of parameters and objects passed between processes.
  - *Kernel:* the message system is implemented in the kernel. Message parameters are checked by the kernel.

- *Context Switching: Choices* provides variable weight context switching. Context switching between one light weight process and another in the same virtual memory space has very little overhead[34]. Context switching between heavy weight processes residing in separate address spaces incurs considerably more overhead. All context switching is implemented in the kernel.

- *Message/Process Interaction:*

  - *Asynchronous*: When a message is sent, the process does not wait for the message to be delivered to the buffer of the receiving process. A copy of the message is made and the kernel returns from the call immediately. When a process attempts to receive a message of a particular type, if the message is not in its port, the receive returns immediately with an *id* that the receiver process can later use to get the message. If the message is in the port the receiver receives the message immediately.
  - *Synchronous*: When a message is sent, the process blocks until the system can send the message to the receiving process. When the receiving process receives a copy of the message the sender is unblocked. When a process does a blocking receive, it waits for the sender to send the message.

- *MessagePort:* this is similar to a mailbox. A message may be left in a mailbox. Typically, a mailbox has one receiver at any particular time and multiple senders. A message port has a queue structure associated with it. The queue may be static or dynamic. All our structures are static. Dynamic memory allocation incurs a large runtime overhead.

- *Transport*:

  - *Process:* an independent process copies the message from source to destination.
  - *Buffered:* the receiver process incurs the overhead of message transfer. When performing a receive the kernel will search the queues for relevant messages.

- *Synchronization*:

  - *Semaphore*: semaphores are implemented in *Choices* by blocking the process and not by busy-waiting.
  - *Spinlock*: this is implemented using the following sequence: 1) read the spinlock variable into the cache, 2) while the spinlock is locked, spin read the variable, and 3) when the value changes try to acquire the lock. If the lock is not acquired, the algorithm is retried. Reading the value into the cache reduces bus contention. A disadvantage of the algorithm is that when the lock is freed, many processes may attempt to acquire the lock causing memory contention[1].

- *Transfer:*

  - *DoubleCopy:* the message is copied into a kernel buffer and then out into a user buffer.
  - *SingleCopy:* the message is copied once from the sender buffer to a receiver buffer in a shared memory region. This is referred to as restricted message passing[40].
  - *PointerTransfer:* buffer pointers are exchanged but data is not copied. One process passes a pointer to a shared data region to another process. The receiver must dereference the pointer to get access to the message data.

3.2 Implementation Issues

In this section, we focus on the queue management alternatives for ports, the memory management scheme for buffers, the user level message system implementation, gang scheduling and the kernel to application interface.

MessagePort Queues In all the implementations of the message passing system, each port has a message queue associated with it. A message for a process may need to be queued up either because the receiver may not be ready to receive the message or the sender may perform an asynchronous send or receive. Each entry in the message queue may have a non-zero data segment associated with it. Concurrent access to the queue by senders and receivers may lead to contention. We describe two solutions to this problem:

1. A sender invokes a *send* which adds a message to the head of the message queue. A send operation acquires the current index of the head of the queue, modulo the length of the queue from an index allocator. The index allocator atomically increments and returns the value of the head of the queue. Each sender acquires a different index, and can proceed to update the queue in parallel. The critical section for the index allocator is extremely small (a few machine instructions).

   The receiver invokes a *receive* operation which takes messages from the tail. The tail pointer is incremented. If the head and tail of the queue are equal (we contend that this is rare) then a sender is blocked until they are no longer equal. This can be avoided by requiring the queue length and number of cooperating processors to be equal, and by disallowing asynchronous sends.

2. Alternatively, previous message queue entries are reused once the messages in them have been received. This method invalidates the cache less often than the circular queue scheme but results in a lot more contention for acquiring a reused queue entry. Each entry requires its own lock.

The first alternative may continuously invalidate the cache, particularly for large messages. The second alternative may cause excessive contention for queue locks. We chose the first method because it was similar to that used by [16, 17] on shared memory architectures. However, our implementation does not require a lock per entry since we only allow one receiver and multiple senders per MessagePort. Multiple receivers per queue would necessitate the use of a lock per queue entry.

Message Buffer The kernel keeps preallocated buffers for the double copy version of the message system. This eliminates the overhead of calling the kernel allocator at message delivery time. For the single copy and pointer message systems data buffer areas are dynamically allocated as part of the application. The allocator for shared user memory is synchronized to permit concurrent operation.

User Level Implementation Recent work shows that interaddress space communication primitives can be implemented very efficiently at user level [5]. Such implementations may be particularly appropriate for applications that are tightly coupled, as is the case of the hypercube applications. In *Choices* we designed and implemented a user level message system library.

The user level message system contains shared data (message queues, spinlock variables) and private data (node identifiers). The user level message system has two types of methods: methods that access shared data and methods that access private data. In order to implement the messaging system, memory allocation primitives are needed to allocate shared memory space and per user private memory space. Since C++ does not provide facilities or support for shared objects, these methods are implemented through programming conventions. C++ provides a private memory operator *new*. We designed a shared memory allocator. During initialization of an application, a pointer is passed to the application containing the location of the user level message system.

The shared allocator uses a section of shared memory that is preallocated. For simplicity, the shared memory is identified as a memory mapped file. The size of the region is chosen a priori to be some large value; if the allocators run out of memory the allocator fails. This approach pre-partitions the address space of each process into shared and unshared regions. The shared region appears in the same virtual memory locations in each virtual address space in order to avoid having to realign pointers.

Gang Scheduling Many parallel applications use processes that have tightly-coupled activities. Often, the processes will synchronize their activities through spinlocks rather than semaphores to avoid the overhead of

blocking on a semaphore. Performance requirements demand that such applications have all their processes running at the same time, one per processor, otherwise a process may spend considerable time waiting on a spin lock that is held by a process that is not running. Hypercube message passing applications are programmed to have tightly coupled activities as message latency through the hypercube network is very small[12]. Such one-to-one process to processor allocation for the duration of an application is called *gang scheduling*.

In *Choices*, each processor in a multiprocessor has a ready queue. Many processors can share the same ready queue and primitives allow the ready queues of processors and processes to change. This is also known as self scheduling in Mach[6]. The ready queues provide an abstraction for both short term and medium term scheduling. There are system, application, and uninterruptible processes and each is implemented to optimize context switch times. Unlike timesharing application processes, uninterruptible application processes run to completion and cannot be preempted or interrupted individually. A special entity called a *Gang* is used as a placeholder for the members of a gang. A Gang is a schedulable entity but it may not be run: instead the gang members are run. When a gang is ready to be dispatched, the scheduler notes the number of gang members. As idle processors attempt to acquire processes to run from the system scheduler, they have their ready queues changed to a special queue called the *GangScheduler*. When the required number of processors have been collected, the gang members are added to the GangScheduler. The processors then take gang members from the GangSheduler one by one.

This approach requires no organizer process to schedule processes on processors. This form of gang scheduling might be termed decentralized gang scheduling. A drawback to the approach is that all the processors access the same GangScheduler simultaneously when the last processor has been collected.

When a process completes, it notifies the GangScheduler. When all the processes in the gang have finished, the GangScheduler changes the ready queues of all the processors back to their old ready queues returning the system to its original configuration.

Application/Kernel interface The Application/Kernel interface is implemented through Proxies. Proxies normally perform a large number of checks when a kernel call is made. For example, a check is made to see if the current domain has the right to use the proxy. There is a check to see if the kernel object referred to by the proxy is valid. We customized the proxy interface to remove many of these checks. This greatly improved the performance of the system.

4 Benchmarks

Although individual timings of the performance of system primitives are useful, in a parallel system with many concurrent activities it is essential to measure the performance of actual applications. We include both forms of measurement.

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 hypercube pseudo-code below shows an implementation of this benchmark as a sequence of messages:

```
Process 1           Process 2
csend(buffer1,P2)   .
.                   crecv(buffer2)
.                   csend(buffer2,P1)
crecv(buffer1)      .
```

We vary this basic test by accessing the data that is sent between processes in the ring. Whether or not the data is referenced or written can modify message latency.

The second benchmark is a parallel version of the Fast Fourier Transform (FFT) [27]. The Fast Fourier Transform is a computational technique that is used extensively in branches of engineering. It has been used to accelerate certain computations. 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 network. The $log_2(P)$ network stage involve 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 $nlog_2(n)$. Figure 2 shows the sizes of messages for the different data set sizes and number of processors used in the benchmark experiments.

Message Size Distribution for FFT					
Numbers procs	128	256	512	**1024**	2048
2	512	1024	2048	**4096**	8192
4	256	512	1024	**2048**	4096
8	128	256	512	**1024**	2048

Figure 2: FFT message data size distribution for varying numbers of processors

The third benchmark is the Simplex linear optimization program. We chose this code because its behavior had been studied extensively on the iPSC/2 by Stunkel [36] on the hypercube distributed memory machines, and because it included non-trivial, data dependent communication patterns.[1]

In our parallel Simplex algorithm, each processor repeatedly finds a minimum among its local data values and then participates in the identification of a global minimum, the pivot element. The processor that contains the minimum value broadcasts a matrix row or column, depending on the data partition chosen, to all the other processors. Thus, the pattern of interprocessor communication is data dependent, the distribution of message sizes is bimodal, and there are many more small messages than large ones.

Simplex Processors versus Messages	
Processors	Messages
2	585
4	881
8	1528

Figure 3: Increasing number of messages as processors increase

Message Size Distribution for Simplex with 2 to 8 nodes	
message size	percentage of total
128	81.99 (pivot selection)
814	18 (pivot data)
1192	0.01
251	weighted average size

Figure 4: Simplex message size distribution for a particular problem with 2-8 processors

Figure 3 shows the number of messages transmitted between processors as the number of processors increases for a data set used in one of the measurements. Figure 4 shows an example distribution of message sizes for

[1] Recall that linear optimization algorithms seek to find a solution to an underconstrained linear system that maximizes (or minimizes) some function subject to a linear constraint.

a data set used in one of the measurements. Such bimodal distributions occur in many networks and have resulted in protocol proposals that are optimized for such distributions[11].

5 Performance Experiments

Our experimental benchmark results can be divided into three sections: message latency, application performance, and support technology overhead. Each of the three sections use an Intel hypercube implementation of the benchmark as a control.

First, we measure message latencies for different implementations of the hypercube message passing scheme using the ring message benchmark. We also present how these results change if applications read or write the message data. Next, we measure the performance of the FFT and Simplex application benchmarks using the different message passing implementations. Third, we provide measurements of the overhead of gang scheduling and process creation, and virtual function calls.

In each of the benchmarking examples, we compare our implementation of the messaging primitives on a shared memory machine with the NX/2 operating system implementation of them running on the Intel iPSC/2 hypercube. Since the benchmarks are not modified to run under *Choices* on the Encore Multimax, the hypercube measurements act as control experiments for our Encore Multimax implementations. We compare message latency, effect of message data access, application performance and message system overhead. The comparison is used as a check to ensure that we have implemented an efficient message passing system.

5.1 Message Latency

The ring benchmark measurements allow comparison of the performance benefits of each of the various design alternatives introduced in section 3.1. In this section, we seek answers to the following questions:

- What is the cost of copying message data between address spaces?
- Is it faster to synchronize using semaphores or spinlocks?
- What is the added cost of having a separate process deliver messages?
- What performance benefits does a user level message system provide as compared to a kernel implemented message system?
- What effect does accessing the message data have on the performance of the system?

Experiments versus Parameters					
Experiment	Location	Transport	Synchronization	Transfer	ContextSwitch
1 $\diamond$	Kernel	Buffered	Spinlock	SingleCopy	Heavy
2 $\bigtriangledown$	Kernel	Buffered	Spinlock	Pointer	Heavy
3 $\otimes$	Kernel	Process	Semaphore	Doublecopy	Heavy
4 $\times$	Kernel	Buffered	Semaphore	Doublecopy	Heavy
5 $\oplus$	Userlevel	Buffered	Spinlock	SingleCopy	Light
6 $\circ$	Kernel	Buffered	Spinlock	Doublecopy	Heavy

Figure 5: Parameters measured in ring experiment

Figure 5 is a representative list of the experiments that we ran using the ring benchmark. Each experiment consists of running the ring benchmark using a message system built from the components named in the table. Figure 6 shows the performance of each experiment in μseconds per trip for different message sizes. The double copying implementation of Experiment 6 most closely resembles the hypercube message passing semantics.

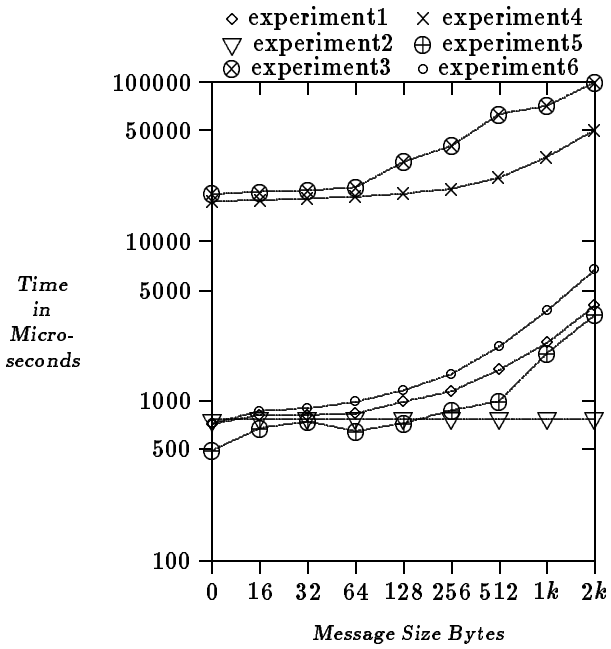

Figure 6: Round trip message times under *Choices* on the Multimax

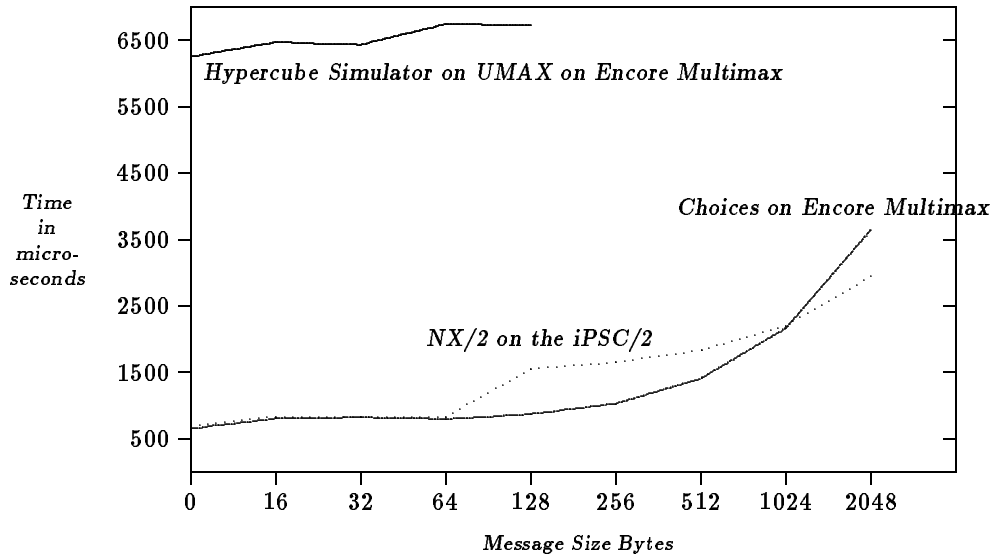

Figure 7: Variation of round trip message times with message size

Figure 7 compares the trip times of Experiment 6 with those of the ring benchmark running under NX/2 on the hypercube. The comparison indicates that the performance of the message system used in Experiment 6 is close to that of the hypercube in terms of absolute performance. In the next sections, we examine these experiments in more detail.

5.1.1 Transfer Semantics

The overhead for transferring message data from one process to another is an important cost in a message system. Three different mechanisms for transferring message data are examined experimentally, each scheme provides slightly different message transfer semantics.

1. *double copy*(represented by Experiment 6): The message is copied into a kernel buffer by the sender process and copied out into a user buffer by the receiver process. This implementation allows hypercube programs to be ported without change to *Choices* on the Encore Multimax. It makes no use of shared memory between users.

2. *single copy*(represented by Experiment 1): The message is copied once into a buffer of the receiver process by the sender process. This is also known as restricted message passing [40]. Buffers are located in shared memory accessible to both the sender and receiver processes.

3. *pointer*(represented by Experiment 2): A message pointer is sent from the sender process to the receiver process. The pointer points to a buffer that is in a shared memory region accessible to both processes.

```
copy
        movd    r3,r0
        addr    16(r0),r0
        movd    r1,r2
        addr    16(r2),r2
        movd    0(r2),0(r0)
read
        movd    r2,r1
        addr    32(r1),r1
        cmpqd   $0,0(r1)
write
        movd    r1,r0
        addr    24(r0),r0
        movqd   $0,0(r0)
```

Figure 8: The assembly code for copy, read and write

Any transfer of data from one process to another involving a copy depends on the rate at which data can be moved from one buffer to another. A copy program for the Encore Multimax consists of a loop that repeatedly reads and writes 4 bytes of data to memory. The Encore Multimax executes approximately two million instructions a second so that a copy could not transfer more than 4 bytes per μsec. The maximum data rate may be decreased by indexing and by loop instructions and by memory contention since the Multimax is a multiprocessor. Thus, a processor performing a copy will use only a fraction of the bandwidth of the Encore Multimax backplane (8 bytes per 80 nanoseconds.)

Figure 9 shows the measured overhead for copying data between buffers on an Encore Multimax. The achieved data transfer rate is approximately two bytes per μseconds. The copy program has the 5 instructions shown in Figure 8. Only 2 of the 5 instructions access memory accounting for the slower than estimated copying rate. Loop instructions are avoided by unrolling the copy loop. The copy data rate has a major impact on the performance of the single and double copy message system implementations.

Results Figure 6 compares the message latency between the three message transfer mechanisms. In the pointer implementation (Experiment 2), message latency remains constant as the message size increases. Both the single copy (Experiment 1) and double copy (Experiment 6) implementations suffer copy overhead and have similar performance behaviors. As is expected, a single copy implementation is faster than a double copy. The difference for a 2048 byte message size is 2690 μseconds, of which almost 2048 μseconds can be accounted for by the extra copying of 4096 bytes of data (round trip copying overhead.)

Analysis In order to explain in further detail the measurements obtained in the roundtrip message benchmark, we examine the operations involved in a message send and receive for each of the transfer mechanisms. As mentioned earlier, message data copying is a major overhead on the Encore Multimax. On the Encore Multimax, access to memory is 5 times slower than access to cache. The operations involved in sending and receiving a message for single and double copy transfers are shown in Figure 10.

In the single copy scheme, a process receives a message by copying it from the sender's buffer into its local buffer. When a write memory operation occurs in an Encore Multimax processor, it writes the data to the cache, shown as a dashed line in Figure 10. The Encore Multimax cache is write-through and the message data is written back to the buffer concurrently with further processor operations. The receiving process sends the message back. This requires the new receiving process to read the data (after a write through has occurred) into its buffer. In one complete cycle of the message, the message data is written from cache to memory and read from memory to cache a total of four times.

Let the notation R_m^i and W_m^i represent a read from and a write to the memory of processor i, respectively. Let R_c^i and W_c^i represent a read from and a write to the cache of processor i, respectively. The symbol [] is used to indicate when the receiver process becomes a sender process. Parentheses are used to delimit concurrent write-throughs.

The message benchmark program corresponds to the following string of operations performed on each item of data in the message for the single copy message system:

$$R_m^2, R_c^1, W_c^1, W_m^1 [] R_m^1, R_c^2, W_c^2, W_m^2$$

Since the buffers in user memory do not change, the write-through does not have to first read the data in the buffer, see section 2.6.

The operations involved in a double copy are shown in Figure 10. A process sends a message by copying it into a kernel buffer. The process receiving the message reads it from this kernel buffer (after write-through has occurred) and writes it to a local buffer in user memory. This process then reads the local buffer again and writes it to a kernel buffer to send the message back to the original process. The original process then reads the kernel buffer and writes it to its local buffer.

Unless the message data is very large, receiving a message primes the cache with the message data. Thus, copying the message data into the kernel buffer does not require reading the data from memory. The write-through to local memory during receipt of a message can occur concurrently.

Let R_m^K and W_m^K represent a read and write to a kernel buffer. The message benchmark program corresponds to the following string of operations performed on each item of data in the message for the double copy message system:

$$R_c^2, W_c^2, W_m^K, R_m^K, R_c^1, W_c^1, (W_m^1) [] R_c^1, W_c^1, W_m^K, R_m^K, R_c^2, W_c^2, (W_m^2)$$

The string operations show that the double copy is slower by extra cache read and write operations. In theory, these operations can be reduced considerably by an effective cache. Figure 6 shows that the difference between the single copy and the double copy is the time to copy the message data twice. It should be possible to calculate more precisely the overhead using the strings of operations and knowledge of the cache. In practice, interference between instruction fetches and data fetches in the cache make such calculations very difficult.

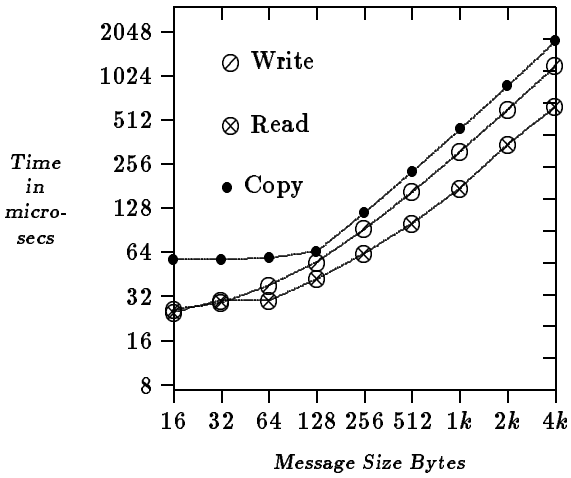

Figure 9: Buffer copy, read and write times on the Encore Multimax

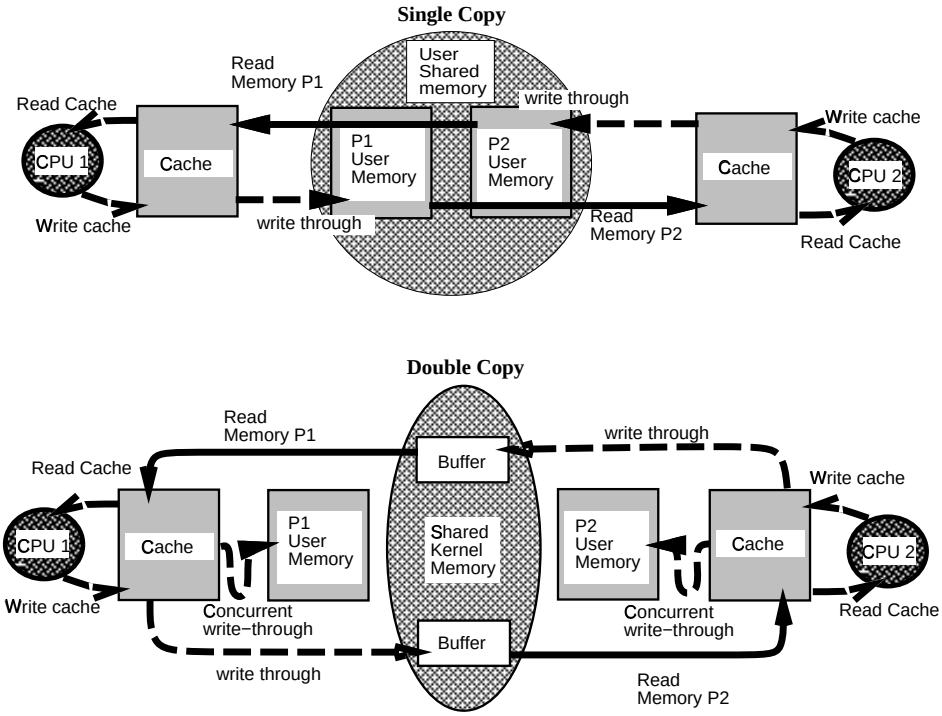

Figure 10: Cache behavior for single and double copy implementations

5.1.2 Analysis of Hypercube Message Latency in the Control Experiment

Despite the Intel hypercube processor being twice as fast as the Encore Multimax processor, Figure 7 shows that the message latency of Experiment 6 under *Choices* is similar to the hypercube for very small message sizes. As the message size grows to 100 bytes, however, the *Choices* message passing begins to perform better. At message sizes past 1024 bytes, the *Choices* messaging passing system performs worse. For example, a 4096 byte message has a round trip message time of 4360 μseconds on the hypercube, see Figure 13. A similar message on the Encore Multimax in Experiment 6 has a round trip message time of 12,171 μseconds, see Figure 12.

The first anomaly occurs because the NX/2 uses a special protocol tailored for short, 100 bytes or less, messages. Above 100 bytes, the protocols change and reserve buffer space at the destination processor by means of additional control messages. As the message size increases above 1024 bytes, the hypercube messaging hardware bandwidth compensates for the overhead of the additional messages and the NX/2 messaging system begins to outperform the *Choices* messaging system.

It is interesting to note that the pointer message system of Experiment 1 outperforms the hypercube messaging system for all message sizes. The round trip times for Experiment 1 are a constant 770 μseconds.

As a further comparison, Figure 7 also shows the performance of an Intel iPSC/2 simulator running the benchmark. The simulator is running under the UMAX operating system on the Encore Multimax. The simulator experiences large overhead since each process is a UNIX process, message passing is simulated through UNIX sockets, context switching occurs, and there is no gang scheduling.

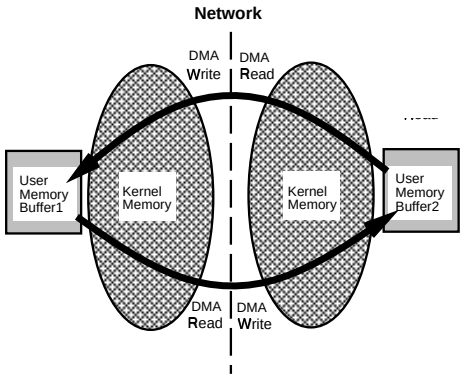

Figure 11: Hypercube behavior for no access message exchange

Figure 11 shows the operations involved in running the ring benchmark between hypercube nodes for messages containing large amounts of data. A message is sent between two hypercube nodes by transferring the message data, under DMA control, from the memory of one node to the memory of another using the hardware messaging support. The message data is not accessed by a processor and is never copied into a cache. The string of operations can be described as:

$$R_m^2, W_m^1 [] R_m^1, W_m^2.$$

5.1.3 Read and Write Access to Message Data

Actual applications rarely receive a message without accessing its contents. We modified the Experiments 1, 2 and 6 to access the message data. In the read variation of the experiments, the message contents are read before being resent. This forces data into the cache, even for the message system using pointers. In the write variation, the message contents are changed before being resent. Figure 12 shows the round trip times measured by the benchmark for Experiments 1, 2, and 6 with no access, read access, and write access to the

message data. For comparison, a similarly modified ring benchmark was run on the hypercube. Figure 13 shows the results measured for the hypercube with no access, read access, and write access. Finally, to account for the behavior of the experiments, we ran a modified copy buffer program on the Encore Multimax to measure the difference between read and write access to a buffer, as shown in Figure 9.

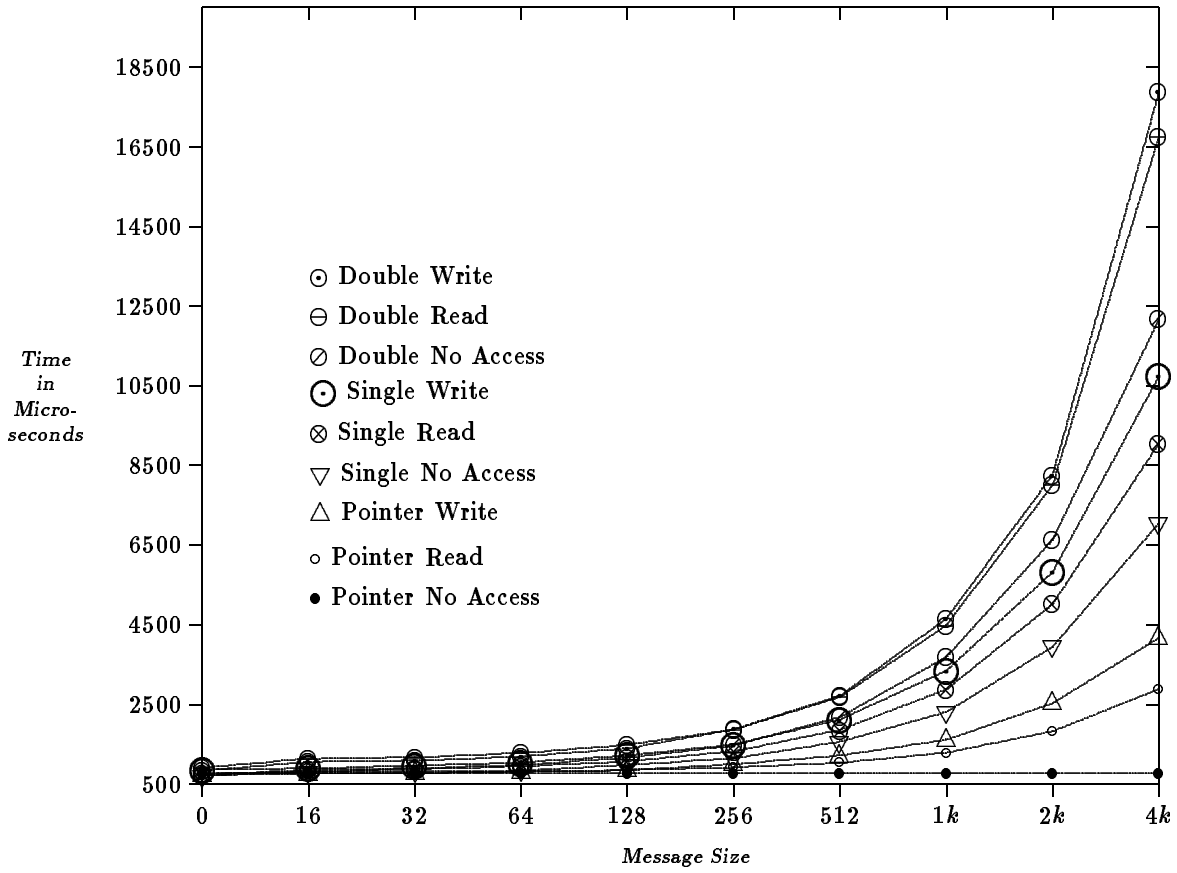

Figure 12: Message passing latency on the *Choices* operating system

Results Figure 12 shows that accessing the data degrades the performance of all the experimental implementations. In particular, the pointer implementation (Experiment 2) no longer has a constant overhead if the contents of the message are accessed. Figure 13 shows similar results for the hypercube. In general, read access performs better than write access. The reason is partially revealed in Figure 9, which shows that reading data from a buffer is faster than writing data to a buffer.

Analysis The difference between no access, read access, and write access can partially be attributed to the overhead of reading or writing a buffer. The instructions used to read or write the buffer are shown in Figure 8. Figure 9 measures the overhead in reading and writing a buffer. For 4096 bytes, a read takes 632 μseconds, a write takes 1190 μseconds, and a copy involving both a read and a write takes 1761 μseconds. These measurements have no loop overhead because the loops are unrolled.

Adding the measured overhead of read or write access to the measured round trip time for a message with no access does not quite account for the round trip times of messages with read or write access. For example, in the pointer experiments the message latency for read access to a message of 4096 bytes is 2884 μseconds. No access message latency for the same size message is 770 μseconds. Reading 8192 bytes of data

(round trip) should take 1264 μseconds. Thus, the measurements account for only 2034 μseconds of the 2883 μseconds latency. We presume that other factors must be involved including interference between the instruction fetches and data fetches in the cache.

The timings for the single copy message system (Experiment 1) and double copy (Experiment 6) are similar to the pointer version. Again, the times cannot be accounted for by a simple computation from our measurements and we presume other factors are slowing down the message system.

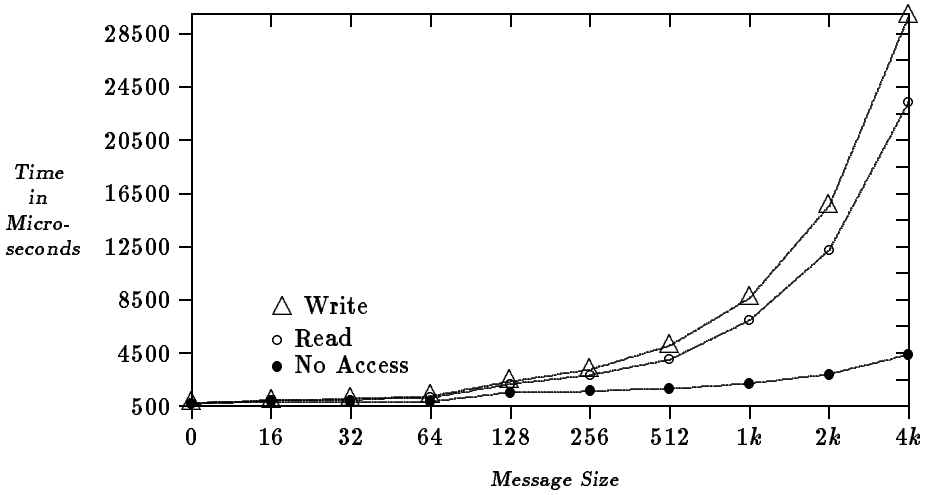

Figure 13: Hypercube message latencies with various types of data accesses

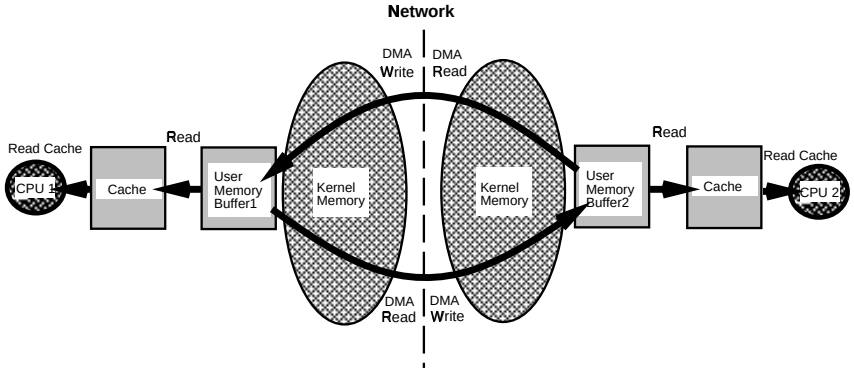

Figure 14: Hypercube behavior for read access message exchange

5.1.4 Analysis of Hypercube Message Latency with Read and Write Access

Figure 13 shows how the hypercube message latencies change when the data in the message is referenced or written. The behavior is remarkably similar to the *Choices* experiments, particularly Experiment 1 using the pointer message system implementation. However, the actual round trip time is *very much larger* when the message data is accessed in the hypercube experiments than in any of the shared memory message experiments shown in Figure 12. For example, a 4096 byte message has a round trip message time of 23,410 μseconds for read access and 29,850 μseconds for write access on the hypercube. The double copy, write

access round trip time is 17,872 μseconds on the Encore Multimax. We hypothesize that poor performance of the iPSC/2 is because it does not have interleaved memory and the processor cannot pipeline data fetches and stores. An alternative hypothesis is that the poor performance is caused because the receiver process no longer preallocates a receive buffer in experiments when access is made to the message. It is not clear to the authors why preallocation would not occur, given the ring benchmark code. When preallocation of a receive buffer does not occur, the behavior of NX/2 is to store any received message in a kernel buffer, see below.

Figure 14 shows the operations involved in running the ring benchmark with read message access between hypercube nodes for messages containing large amounts of data. The message data is transferred under DMA control across the hypercube network. To access the data, the CPU must read the data into its cache. In this model, we have the following string of operations:

$$R_m^2, W_m^1, R_m^1, R_c^1 [] R_m^1, W_m^2, R_m^2, R_c^2.$$

If the message is write accessed, then the message is first read into the cache, changed and then written out with the write-through cache trickling the effects into main memory.

The worst performance occurs on the hypercube with write access when a message is sent but the receiver is not ready to receive the message. In this case, the message is stored within the kernel of the receiver, as shown in Figure 15. This is similar to the double copy message transfer mechanism of Experiment 6. The string of actual memory operations is:

$$R_m^2, W_m^{1k}, R_m^{1k}, R_c^1, W_c^1, (W_m^1) [] R_m^1, W_m^{2k}, R_m^{2k}, R_c^2, W_c^2, (W_m^2).$$

The superscript ik refers to a message stored in the kernel network buffers of processor i. There are 6 sequential memory reads and writes, as opposed to 4 in the double copy case on the Encore Multimax.

5.1.5 Synchronization

The synchronization experiment compares spinlocks (Experiment 6) against semaphores (Experiment 4) while keeping other message system parameters the same, as shown in Figure 5.

Results Figure 6, shows that this change increases the message passing latency by a factor of 10.

Analysis In *Choices* a process performing a P operation on a semaphore may block. The blocked process is put on a semaphore queue. When a V operation occurs, the first blocked process on the semaphore queue is transferred to the ready queue. In the experiment, *Choices* uses a single ready queue. Processes may run on any processor. A process context switch results in the invalidation of the translation lookaside buffer, page tables and the data cache. A rescheduled process may have to refill the cache with appropriate data. Our experiments show that context switching overhead causes message latency to degrade by a factor of 10 for the ring benchmark.

Two similar studies combining theory and simulation have been performed by [29] and [28]. The work of Ousterhout[29] shows that spinning should be chosen over blocking when the waiting time at the point of synchronization is less than two context switch times[29]. A separate study by Mogul [28] has shown that context switching invalidates large portions of a data cache.

The *Choices* context switch times [34] are 412 μsec on the Encore Multimax for application processes that perform floating point operations. Experiment 6, the spinlock implementation of the message system, indicates that a small message takes about 800 μseconds for a round trip. Using a semaphore implementation, a round trip causes two context switches. Therefore, we should choose spinning over blocking as is confirmed by the Experiment 3 measurement shown in Figure 6. For large messages, Experiment 6 has a round trip time that is far in excess of 812 μsec. Most of the overhead is in copying the message data. In Experiment 4, a process will block waiting for this copying to complete. As can be seen from the results in Figure 6, for this unusual case the performance of the semaphore implementation does not improve radically as the waiting times increase.

5.1.6 Transport

The process transport experiment compares using an independent process to transport a message from sender process to receiver process (Experiment 3) against having the sender and receiver process perform the task (Experiment 4). The other message system parameters are the same, as shown in Figure 5.

Results Figure 6 shows that the use of a separate process to transport a message degrades the performance of the message system. The effect increases as the size of the message data increases.

Analysis Adding a separate process to deliver messages introduces another process that must be scheduled. This therefore has the potential of increasing the time lost due to cache invalidation and context switching. If one of the application processes is context switched and the message delivery process is allocated to its processor, the cache will be invalidated. However, when the receiver and sender process cooperate to deliver the message as in the double copy system of Experiment 4, they can both exploit the contents of their caches to improve message transmission times.

5.1.7 Kernel Protection

The kernel/user message system placement experiment compares a kernel implementation (Experiment 1) against a user level implementation (Experiment 5) while keeping most other message system parameters the same, as shown in Figure 5. Since context switching does not occur using spinlocks, the "weight" of the context switch is irrelevant.

Results Figure 6 shows that kernel protection for the message system adds a fixed overhead to message latency that does not vary with the message data size.

Analysis Putting the message system in the kernel adds approximately 144 μseconds to message latency, corresponding to the overhead to make 2 kernel calls. Parameter checking and trap overhead remain a constant as the message size increases.

5.1.8 General Comments

As long as message exchange is synchronized by spinlocks and processes are gang scheduled, it is relatively easy to move a messaging system into and out of a kernel. For applications that send large messages, the overhead caused by kernel access is a small percentage of the round trip time. Placing the message system in the kernel aids program porting because protected access to the system simplifies debugging. For applications that send very small messages very often, moving the messaging system into user space may have major performance benefits.

In this exposition of different messaging techniques, we have not attempted to produce the optimal messaging system but rather measure the impact of various messaging parameters. The results of Experiment 2 could be improved by combining its pointer transfer mechanism with the placement of the message system in user shared-memory, as in Experiment 5. This would reduce the 770 μseconds message latency of Experiment 2 by the constant 144 μseconds. Unfortunately, the advantage of this combination is lost if messages are large and the data in the message are accessed.

The combination of parameters listed as Experiment 6 provide the most accurate reproduction of the NX/2 messaging system. A summary of the breakdown of the round trip time for a null message for Experiment 6 is shown in Figure 16. The component contributing the largest overhead to the round trip time is the kernel call identified in the table as a "proxy call." This call includes the time to trap into the kernel and copy arguments onto the kernel stack. Moving the message system into user space eliminates this cost. Process identification overhead is extremely high as we do not cache process identifiers in memory. The bookkeeping time refers to the time the message system takes to set flags and any process idle or waiting

time for spinlocks. The overhead of programming the message system in C++ corresponds to the virtual function call overhead. Virtual function calls contribute a constant 11.25% to the message latency overhead that is independent of the message data size. Replacing the virtual function calls by C function calls halves the overhead from 72 μseconds to 36 μseconds. Thus, the additional cost for programming the message system in C++ is less than 6%. As the message size increases, the use of C++ becomes an increasingly smaller concern.

In the next sections, we will be discussing message systems 1, 2, and 6. We do not consider message passing systems 3 and 4 because they are too slow and we do not consider 5 because we are interested in retaining some degree of protection between applications and we would like to run distributed memory applications with as few changes as possible.

5.2 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 applications on three versions of the message system: single copy, pointer, and double copy implementations. The three versions were benchmarked in the previous section on message latency as Experiments 1, 2, and 6.

The three implementations differ only in the transfer semantics used for message data. However, as we observed in the last section, this parameter has a significant effect on message latency and should impact application performance. In particular, Figure 6 indicates the pointer version should improve the performance of applications more than either the single or double copy versions. In the next two sections, we present the results of running two applications, the Fast Fourier Transform (FFT) and Simplex algorithms, on the experimental message passing systems.

5.2.1 FFT Application

The FFT application permits a study of how message systems behave as message sizes and numbers increase. The Fast Fourier Transform[27] application was compiled without change and run under *Choices* using the double copy message system that most closely resembles the hypercube message passing semantics. The FFT application had to be rewritten to use the pointer and single copy message passing systems. An effort was made to keep the changes to a minimum and very little code, apart from memory allocation, was changed. In the reimplementations of the application, the data structures that are passed as data in messages are allocated out of shared memory rather than private memory.

Figure 17 contains plots of the speedup versus the number of processors for each implementation of the FFT application for a fixed data size. For comparison purposes, the performance of the FFT on the Intel hypercube under NX/2 is also displayed.

Results Figure 18 is a table comparing the execution times of the three implementations for a varying number of processors. The results are scaled relative to the performance of the pointer implementation. The FFT speedup curves for the three variations in message system transfer semantics are identical for 1 processor, indicating that there is no overhead incurred as a consequence of modifying the FFT algorithm to accommodate the changes in transfer semantics. The pointer implementation performs worse than either of the other implementations when the number of processors is greater than 4. Figure 17 shows that the single copy implementation speeds up better than the other implementations, including the hypercube implementation. The 8 processor case seems to indicate the single copy implementation continues to be a better implementation as the number of processors increases. The speedup curves of the application on the Intel hypercube and the double copy shared memory machine implementation are almost identical.

Analysis The pointer implementation's poor FFT behavior is not predicted by the message latency benchmark results and requires explanation. Examining the FFT program, we observe that the data space for

each processor is allocated as a integral multiple of 64k bytes. The FFT algorithm exchanges messages in a butterfly configuration. Because of this symmetry, the processors in the algorithm access data that are located at addresses that are modulo 64k apart and this results in cache line invalidation for each processor for each data item. Each processor is initialized with a copy of the data set which is 16k bytes in size and is aligned on a 64k boundary. The algorithm on each processor receives messages containing pointers to other processor's data regions. The processor uses the same offsets into these data regions in its computations causing comprehensive cache line invalidation. The measured difference in performance between the pointer and single copy implementations is 12,579 μsecs. If a cache entry is invalidated, it will cause a read from memory when it is accessed. The cache access time is 25 nanoseconds but memory access time is 320 nanoseconds. It is thus very likely that the performance degradation of the pointer implementation can be accounted for by cache line collisions. As the number of processors increases, the number of collisions will increase.

Because the processors are executing symmetric instructions, there is also a possibility that when the cache entry is reread, there will be memory module contention. Processors in an Encore Multimax can access a single memory module every 320 nanoseconds, see section 2.6. In the pointer implementation, four processors can access the same module in 1.28 μsecs. Using the data from Figure 9 measurements, a single processor can read 4 bytes in 617 nanoseconds. If four processors in using the pointer implementation read all of the data in a message in a synchronized manner, contention will effectively halve the performance of reading the message data.

On the other hand, both the single and double copy message systems rearrange the data by copying it to new locations, avoiding much of the cache line invalidation problems. The extra copying overhead in the double copy implementation makes this implementation perform worse than the single copy.

The FFT Hypercube Control Experiments Results Figure 17 shows that the speedup curve of the hypercube implementation of the FFT is similar to the speedup of the double copy Encore Multimax implementation. Instrumenting the hypercube hardware and Encore Multimax software allows a comparison to be made of application and system overhead. Figure 19 shows that the proportion of time spent in the NX/2 messaging system for the FFT application is almost the same as that spent in the double copy messaging system on the Encore for varying FFT data set sizes. This confirms earlier results that the overheads for processing a message of a given size using NX/2 and the double copy message systems are similar. In terms of absolute timings, our measurements showed that the hypercube implementation of the FFT has much better performance than the Encore Multimax implementation because the application benefits from faster floating point hardware on the Intel iPSC/2.

FFT Hypercube Analysis A detailed analysis of the FFT execution traces on the hypercube reveal that there is a variation between the times of successive interprocessor communications. 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 on both systems.

Simplex Application The simplex application is a non-trivial application and permits a study of how message systems behave for bimodal message size distributions and data dependent communication patterns. The behavior of the simplex code has been studied extensively on the iPSC/2 by Stunkel [36]. Again, the simplex application did not have to be changed for the double copy message system, but did require minor changes for the pointer and single copy message systems.

Results Figure 20 shows the speedup of the simplex application on the various message passing systems including the hypercube. The speedup curves of all the message system implementations are very similar but the ordering of the performance of the implementations changes from that for the FFT. The single copy implementation performs worse than either the pointer or double copy implementations for the numbers of

processors shown. The hypercube implementation performs better than any of the shared memory implementations as the number of processors increase. Figure 21 is a table comparing the execution times of the three implementations for a varying number of processors. The results are scaled relative to the performance of the pointer implementation.

Analysis Figure 4 shows that small messages predominate. The weighted average message size for Simplex is 251 bytes. The fixed overhead for transmitting a message in Simplex is large compared with the variable overhead for copying message data. This has the effect of reducing the differences between the various shared memory message passing systems.

In the Simplex application, pivot distribution is broadcast from one processor to the others. The single copy message system is inefficient at broadcasts because each process receiving a message copies the message data. This causes read access memory contention. The double copy message system is more efficient than the single copy one because the sender process copies the message data to be broadcast from its cache into kernel buffers destined for other processes. These other processes copy the data from their individual kernel buffers and avoid memory contention.

The Simplex Hypercube Control Experiments Results The behavior of the shared memory implementations is similar to the NX/2 hypercube implementation. In both cases, the cause of sub-linear speedup is communication overhead.

Analysis Figure 22 shows the fraction of time spent in the message passing system when the Simplex code executes under NX/2 on the Intel iPSC/2 hypercube and under *Choices* using the double copy message system on the Encore Multimax. As the number of processors increases, the fraction of the time spent in the message passing system increases. There are two reasons for this. First, the total computation is fixed, and as the number of processors increases, the fraction of the work assigned to each processor decreases. Second, the total number of message sent by all processors increases; most of these messages are small and represent the overhead to find a global minimum among a larger group of processors. Thus, in the case of the Simplex application, because of its large number of small messages it may be desirable to use a message system in user space.

5.3 Gang Scheduling

Gang scheduling allows a group of N processes to have exclusive access to N processors. Mach researchers[6] identified the following three properties as important gang scheduling parameters:

1. The time to create a gang and add N members to it.
2. The time to schedule a gang and collect N processors.
3. The time to dispatch N processes on N processors.

We extend the work of the Mach researchers by investigating how these parameters vary with the number of processors and how some of the parameters vary with multiprogramming workloads. Figure 23 presents the values for the above parameters in the absence of any multiprogrammed workload. Performance varies according to the size of the gang. The performance compares favorably with Mach[6], although the two systems are very different making comparisons difficult. The performance of *Choices* could further be improved by reducing contention for the shared *GangScheduler* by duplicating the *GangScheduler* for every processor.

Analysis Figure 23 shows that parameter 1, the time to create a gang and add members to it, varies slightly as the number of members increases. The cost to add members to the gang increases linearly and is the cost to sequentially access the gang member list structure. Parameter 2 has two components. A fixed overhead for scheduling the gang that does not change with the number of gangs members and a variable component for collecting the N processors for the N gang members. This number increases linearly with the number of gang members. These numbers were gathered in the absence of multiprogramming. The results for parameter 3, shows that the time taken to run processes that are ready is larger than the corresponding uniprocessor dispatch time (2.0 milliseconds) and increases with the number of processors.

The performance of collecting a gang of processes in a timeshared, multiprogrammed environment is highly dependent upon the multiprogramming workload and timeslice, unless preemption is permitted. In a timeshared environment, the time-slice quantum ensures a bound can be placed on the time it will take for the processors to be collected for the gang. In *Choices*, dispatching a gang is decentralized and only involves collected processors. Once the processors are collected, the time to dispatch the processes should not depend on workload. Figure 24 shows how the time to collect all the processors for a gang varies with the number of gang members p. These numbers are more than those in Figure 23, because the measurements were taken in a multiprogrammed environment.

The theoretical time to collect a gang of size p when there are N processors in the system and the time slice is TS seconds can be estimated. The time to collect the first processor, $First$, is $TS/(2 \times N)$ if the timeslicing interrupts of the processes are distributed uniformly and independently between 0-TS seconds. The first processor runs the process that collects the processors, reducing the problem to collecting $p-1$ out of $N-1$ processors. Let $Collect$ be the time for the first process to collect the remaining $p-1$ processors. Each of the $p-1$ processors needed for the collection can be chosen out of the next $p-1$ processors that are time-sliced. Assuming the remaining $N-1$ processor time-slice interrupts are independent of one another, each collected processor contributes a waiting time of $TS/(2 \times (N-1))$ to the collection time. There are $p-1$ processors being collected. This leads to a total time and time to start collecting of:

$$Collect(TS, N, p) = (TS \times (p-1))/(2 \times (N-1))$$

$$First(TS, N) = TS/(2 \times N).$$

We simulated a multiprogrammed workload by flooding all 16 processors of the Encore Multimax with randomly scheduled processes. The times for the collection of p processors were measured. The timeslice was set to 10 milliseconds. Figure 24 shows the expected and the measured times for collecting processes. $Collect(TS, N, p)$ predicts the measurements made fairly accurately. However, as the number of collected processors increase, the accuracy of the estimated collection times decreases. Each time-slice interrupt incurs a slight overhead that is not accounted for by the estimated collection time. As the number of processes increases, the sum of these overheads increase linearly. The results are averaged over 10 experiments to eliminate small perturbations.

6 Conclusion

In this paper, we experimentally study the comparative performance of different message passing system implementations on a shared memory multiprocessor using benchmarks and applications. The message passing system is based on the NX/2 hypercube message passing primitives and this allows us to compare the benchmarks and applications on both machines. The comparison acts as a control to ensure that we have implemented the message system efficiently.

All of the software written for the experiment is object-oriented and written in C++including the message system implementations and the operating system. Object-oriented techniques were used to structure the message system to minimize the coding differences between different message system implementations. The message system design alternatives considered were buffering, buffer and queue organization, reference and value semantics, synchronization, coordination strategy, and the location of the system in user or kernel space.

The results from our study show that different message system implementations can impact the performance of benchmarks and applications considerably. In our experiments, a message system that uses a blocking synchronization primitive ran applications an order of magnitude slower than one that used spin-locks and gang scheduling. Kernel implementations of a message system imposed a constant additional overhead on application performance caused by kernel entry and exit costs. The FFT application sends large messages and a kernel implementation would not impose a significant overhead. However, the Simplex application sends many small messages and a user-level implementation would improve its performance.

In some applications, a message system based on passing pointers to shared memory locations can be very efficient. In other applications, like the FFT application we studied, pointer passing schemes may result in cache line invalidation and memory contention as the application performs its parallel computation. Message systems based on a single copy or double copy of the message data can perform well for such applications by moving the data to locations where cache line invalidation is reduced and memory contention is eliminated. Message systems based on a double copy scheme impose much overhead on simple applications. In the FFT application the overhead makes the double copy scheme significantly slower than the single copy scheme.

When there are a large number of small messages in an application the mechanism used to transfer the data is not as relevant as the fixed cost of sending and receiving messages. The Simplex application has a large number of small messages and the behavior of the different kernel messaging systems was similar. Memory contention appears to be reduced for broadcasts of a message from one process to others using a double copy message system instead of a single copy or pointer implementation. This effect was noticed in the Simplex algorithm in which one processor broadcasts a pivot point message to all other the other processors.

We have also investigated the effects of other parts of the operating system that support message based applications: gang scheduling and virtual functions. Our gang scheduling numbers are close to those reported by Black [6] for the Mach operating system. We have also presented the results demonstrating the effect of multiprogramming on various gang scheduling parameters. Our work shows that the overhead of virtual functions is small for null messages and negligible for larger message sizes.

Our recommendations for writing a message system for a shared memory multiprocessor are:

1. It is very unlikely that a particular message system will work efficiently for a large number of different applications. Design the messaging system so that it can be customized for an application.

2. Eliminate context switching by using gang scheduling and non preemptable processes. Context switching is a major overhead on the machines on which we ran our tests. RISC based architectures are likely to make context switching worse.

3. Remove blocking synchronization primitives. For the same reason, synchronization operations which cause a context switch are a major overhead.

4. Organize message queues to eliminate lock contention. Memory contention in the Encore Multimax is a real issue, even for quite small numbers of processors if those processors are trying to access the same lock.

5. Choose an appropriate message transfer semantics for the problem. Some message transfer mechanisms are too heavy weight for some applications that do not require a particular constrained message semantics. Other message semantics are too light weight for applications and create memory contention.

   (a) Eliminate copying if the message data is not accessed. A pointer transfer message system is quite adequate if the data in a message is not to be accessed directly by many nodes that the message passes through.

   (b) Choose a copy transfer semantics to eliminate memory contention, where necessary. A pointer transfer message system suffers severely from memory contention if the applications are symmetric and use broadcasts. Copy transfer semantics distributes memory accesses, reducing cache line invalidation and memory contention for the same memory location and memory module.

(c) Choose address locations to reduce instruction and data interference in the cache. It is very difficult to develop a message system that avoids instruction and data cache interference.

(d) Optimize the way the cache is used in the transfer mechanism for particular application messaging patterns. Implement broadcasts by copying from a cached copy of the message if possible.

6. Use a kernel-based message system to simplify the porting or development of applications to a multiprocessor. A kernel-based message system provides a better debugging environment.

7. Use a user-level message system for applications that send lots of small messages.

8. Use object-oriented programming techniques to simplify customizing a message system for a particular application. The techniques introduce virtual function method lookup overhead, but the overhead is much smaller than the benefits obtained by customization.

In general, our research supports the belief that an object-oriented operating system like *Choices* provides an excellent testbed for parallel operating system studies. We found small benchmark programs to be inadequate to measure or predict application performance. Real applications must be used to validate parallel algorithm design decisions. Finally, our results indicate that one message system implementation is not optimal to support a wide range of useful applications efficiently.

7 Acknowledgements

We would like to thank Dan Reed for invaluable comments on many drafts of this paper. He also provided us with the code for the Simplex and FFT algorithms. He gave us access to an instrumented version of the NX/2 operating system that was used to obtain message passing overhead times on the Intel iPSC/2 hypercube. We would also like to thank Anthony Reeves of Cornell University since the message system was originally built to support the port of his Paragon parallel programming environment to *Choices*.

References

[1] T.E. Anderson. The Performance of Spinlocking Alternatives for Shared Memory Multiprocessors. In *IEEE Transactions on Parallel and Distributed Processing*, pages 6–16, January 1990.

[2] Thomas Anderson, Edward Lazowska, and Henry Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory multiprocessors. In *IEEE Transactions on Computers*, pages 1631–1644. IEEE, 1989.

[3] Forest Baskett, J. H. Howard, and John T. Montague. Task Communication in DEMOS. *ACM Operating Systems Review*, pages 23–31, November 1977.

[4] Brain Bershad, T.E. Anderson, Ed Lazowska, and Henry Levy. Light Weight Remote Procedure Call. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 102–112, December 1989.

[5] Brian Bershad, T. Anderson, Edward Lazowska, and Henry Levy. User Level Interprocess Communication for Shared Memory Multiprocessors. Technical Report 90-05-07, University of Washington, July 1990.

[6] David Black. Scheduling support for concurrency and parallelism in the Mach Operating system. *COMPUTER*, pages 35–43, 1990.

[7] David Bradley. First and second generation hypercube performance. Technical Report UIUCDCS-R-88-1455, University of Illinois Urbana-Champaign, September 1988.

[8] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[9] F.G. Chase J.S., Amador, Levy HM Lazowska E.D., and Littlefield RJ. The Amber Sytem: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 28–50, December 1989.

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

[11] David Cheriton and Carey Williamson. Network Measurement of the VMTP Request-Response Protocol in the V Distributed System. *Proceedings of the 1987 ACM Sigmetrics Conference*, pages 128–140, 1987.

[12] Paul Close. The iPSC/2 Node Architecture. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, January 1986.

[13] Encore Computer Corp. *Multimax Technical Summary.* Encore, Marlborough, Massachusetts, 1986.

[14] P. Dasgupta, R. J. LeBlanc, and W. F. Appelbe. The Clouds Distributed Operating System: functional description, implementation details and related work. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9. IEEE, June 1988.

[15] Peter Deutsch. *Design Reuse and Frameworks in the Smalltalk-80 Programming system.* ACM Press, Cambridge,Mass, 1989.

[16] Jeffrey Goodman, Mary Vernon, and Philip Voest. Effficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–73. ACM, 1989.

[17] Gottlieb, Lubachevsky, and Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequentila processors. In *ACM Transactions on Programming Languages and Systems*, pages 164–189. ACM, 1983.

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

[19] Nayeem Islam and Roy Campbell. An Object-Oriented Framework for Message Passing. Technical Report UIUCDCS-R-91-, University of Illinois Urbana-Champaign, November 1991.

[20] Herb Jacobs. A User-tunable Multiple Processor Scheduler. In *1986 Winter USENIX Conference Proceedings*, pages 183–191, January 1986.

[21] Gary M. Johnston and Roy H. Campbell. An Object-Oriented Implementation of Distributed Virtual Memory. In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 39–57, Ft. Lauderdale, Florida, October 1989.

[22] T.J. LeBlanc. Shared Memory versus Message passing in a tightly coupled multiprocessor:A case study. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466. ACM and IEEE, 1986.

[23] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[24] Calvin Lin and Lawrence Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In *Proceedings of the 1990 International conference on parallel processing*, pages 163–170, August 1990.

[25] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference*, pages 65–79, Denver, Colorado, October 1988.

[26] Allen D. Malony, Daniel A. Reed, and David C. Rudolph. Integrating Performance Data Collection, Analysis, and Visualization. In Margaret Simmons, Rebecca Koskela, and Ingrid Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization.* Addison-Wesley Publishing Company, 1990.

[27] 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.

[28] Jeffrey Mogul and Anita Borg. The Effect of Context Swithces on Cache Performance. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84. ACM, 1991.

[29] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International conference on Distributed Computing Systems*, pages 22–30, July 1982.

[30] Paul Pierce. The NX/2 Operating System. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, January 1986.

[31] Richard Rashid. From RIG to Accent to Mach: the Evolution of a Network Operating System. *Proceedings of the ACM/IEEE 8th Compuity Society Fall Joint Conference*, pages 314–334, 1981.

[32] Daniel Reed and Richard Fujimoto. *Multicomputer Networks - Message based Parallel Processing.* The MIT Press, Cambridge, Massachusetts, 1986.

[33] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[34] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, pages 103–114, San Francisco, California, April 1990.

[35] Marc Shapiro. The Design of a Distributed Object-Oriented Operating System for Office Applications. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), November 1988.

[36] Craig B. Stunkel. Linear optimization via message-based parallel processing. In *Journal of Parallel and Distributed Computing*, pages 264–271, August 1988.

[37] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *ACM Operating Systems Review*, 15(3):51–64, July 1981.

[38] Marvin Theimer. *Preemptable Remote Execution Facilities for Loosely Cooupled Distributed Systems* . PhD thesis, Stanford, June 1986.

[39] A. Tucker and A. Gupta. Process Control and Scheduling issues for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 159–166. ACM, 1989.

[40] S-Y Tzou and David Anderson. A Performance Evaluation of the Dash Message Passing System. In *Software Practice and Experience*, pages 1631–1644. John Wiley, 1991.

[41] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying Current Research in Object-Oriented Programming. *Communications of the ACM*, 33(9):104–124, September 1990.

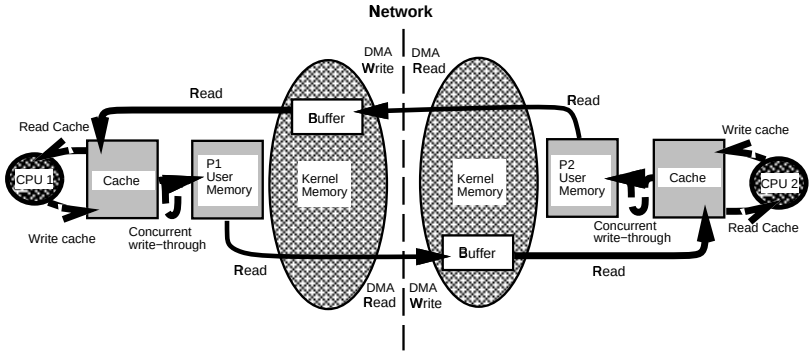

Figure 15: Hypercube behavior for write access message exchange

Round trip message times for the Ring Program			
Number	Activity	Overhead	Total
4	Proxy calls	72μsec	288μsec
4	Process identifications	30μsec	120μsec
12	Virtual functions	6μsec	72μsec
4	Book keeping	40μsec	160 μsec
	TOTAL		640μsec

Figure 16: Overhead for null message

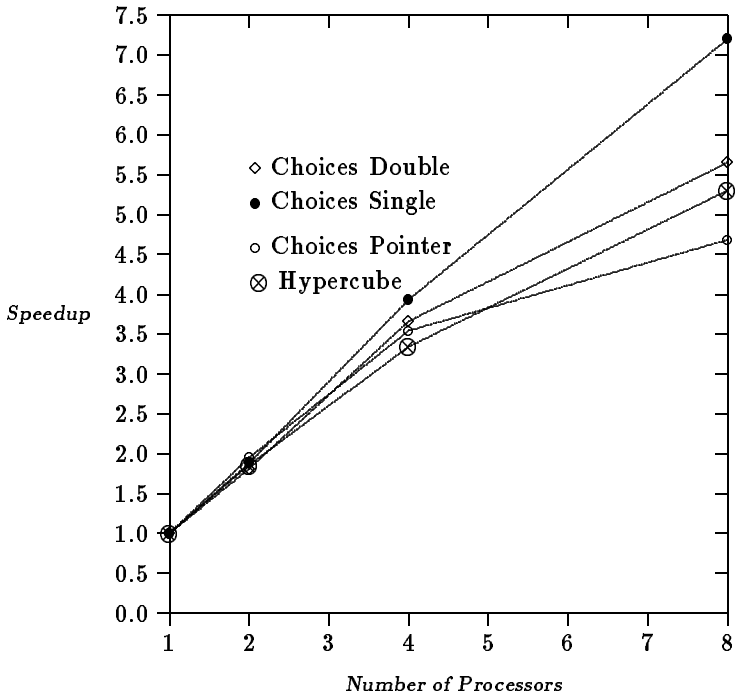

Figure 17: Speedup of the FFT algorithm on *Choices* and on the NX/2 for N= 1024

Cross Comparison of FFT on various implementations			
Number of Nodes	Pointer	SingleCopy	DoubleCopy
1	1	1	1
2	1	1.08	1.04
4	1	0.90	0.96
8	1	0.65	0.83

Figure 18: Cross Comparison of FFT with number of processors N = 1024

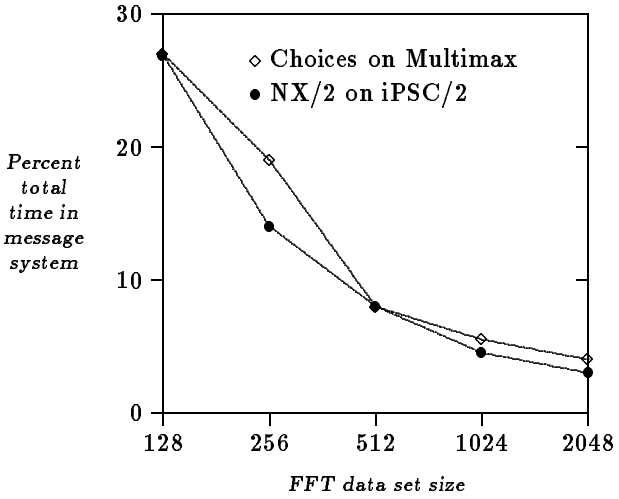

Figure 19: Variation of percentage of time in message system with incr easing data size for an FFT running on 4 processors

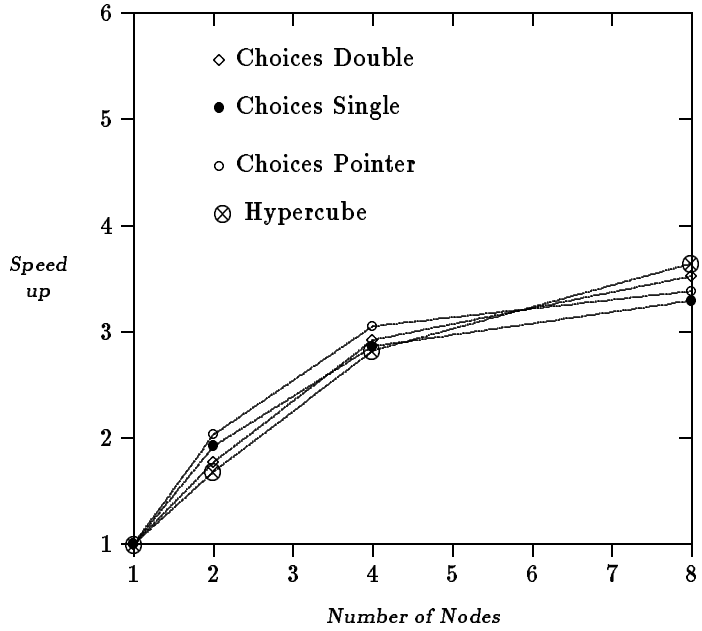

Figure 20: Speedup of the Simplex algorithm on Choices and on NX/2

Cross Comparison of Simplex on Choices			
Number of Nodes	Pointer	SingleCopy	DoubleCopy
1	1	1	1
2	1	1.06	1.20
4	1	1.03	1.05
8	1	0.98	0.95

Figure 21: Cross comparison of Simplex with number of processors

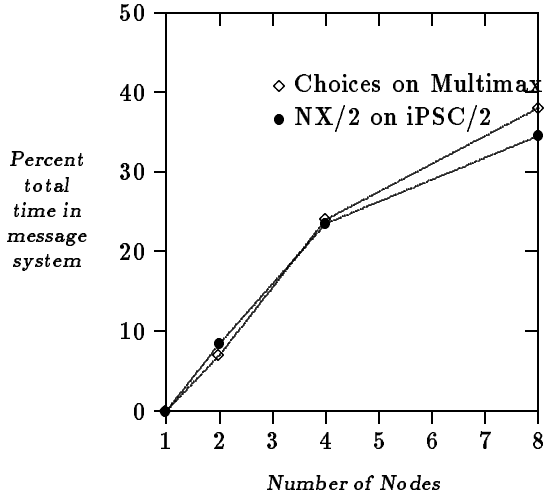

Figure 22: Percentage of time spent in the message system for the Simplex algorithm on NX/2 and *Choices*

Gang Scheduling Overhead in Milliseconds				
Number of gang members	n =2	n=4	n=6	n=8
1. Create gang and add to list	78	85	90	98
2. Schedule gang and collect processors	4.0	4.2	4.4	5.0
3. (a)Ready to running(min)	3.0	3.1	3.8	4.3
3. (b) Ready to running (max)	3.4	3.5	4.6	5.0

Figure 23: Gang scheduling parameters

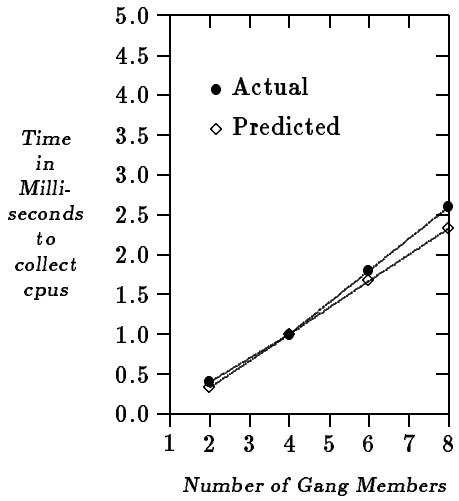

Figure 24: Time to collect processors under multiprogrammed workloads