Direct Deposit – When Message Passing meets Shared Memory

Thomas M. Stricker

May 19, 2000

CMU-CS-96-166
CMU-CS-00-133 (REV)

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by DARPA/CMO to Carnegie Mellon University.

Keywords: Parallel compilers, high performance Fortran, direct deposit message passing, deposit model, postal-model, rendezvous-model, message passing, shared memory, remote store, remote load, memory system merformance, massively parallel multiprocessors, Cray T3E, Cray T3D, Intel iWarp, Intel Paragon

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This thesis is motivated by a number of massively parallel supercomputers with a distributed memory architecture that were developed in the early 1990ies and introduced commercially to the market around the middle of the decade. The most notable introductions were the MassPar, the CM5 by Thinking Machines [22, 71], the N-cube machines [78], the iWarp by CMU and Intel [47], the Paragon Series by Intel [20], the SP-1 by IBM [92], the KSR by Kendall Square Research [43], the T3D and the T3E from Cray Research [1, 90]

Upon their introduction all those machines claimed to be scalable to a large number of processor and to offer an unprecedented compute power at a much lower cost than conventional vector supercomputers. The rate of innovations for interconnect network technologies, network interface chips and communication hardware support in the node architecture was extremely high and that made that field very exiting. It seemed that only one issue could limit the success of these machines on the market - a good program model that allow software to be portable and reused. Unlike for vector supercomputers, MPP were lacking conceptionally clean, convenient and standardized programming model.

There are two basic programming models for writing parallel and distributed programs: *message passing distributed memory* and *coherent shared memory*. In the 1990ies both models held strong camps, well defended by their promoters. Fully coherent shared memory seemed to be more desirable, but much harder to implement and therefore many systems architects were involved in building scalable shared memory machines with full cache coherency. This research lead to several prototypes at Stanford univerity and elsewhere and finally lead to the commercial Origin product series by Silicon Graphics. The message passing camp comprised the pragmatic architecture people and the visionary software architects, trying to get the best out of some existing hardware, while offering simple library functions to the programmers and working on the standardization of the libraries. Writing application code for message passing systems the programmers must deal with many aspects of parallel programming themselves, but the there are few limits to manual optimization and no bounds on a most efficient use of networks.

Looking from a distance at the two traditional camps of parallel and distributed computing a third view became apparent to me. Maybe the platforms of the future have to be programmed at a higher level and compilers or parallel programming generators will be used extensively at some time in the future. With such a perspective, it might be irrelevant if the machine belongs to the fully coherent shared memory system cathegory or to the message passing distributed memort cathegory and the most important thing will be that a machine can move the data most efficiently between any two different processing elements. I also discovered that the complexity and the efficiency of the underlying system software has a great influence on the sustained performance of the entire system with hard- and software included. So making things simple and fast might actually win over following a strict philosophy to its pure form into all its complexities.

During the later 1990ies another trend in the hardware seemed to support this philosophy. More and more communication systems offered data transfer mechanisms, that could be highly useful in the shared memory or the message passing world. Fully coherent shared memory machines started to experiment with programmable, customizable protocol coprocesor, making part of the shared memory communication explicit and the message passing world move towards fine grain remote load/store that could operate in the global address space of an entire machine (without much coherence and subject to unusual cost models of course far from a simple non uniform memory access (NUMA)).

Last but not least, this thesis is motivated by my participation in the iWarp project, one of the largest systems building effort ever undertaken as contract research by a university. In many respects the hardware of the iWarp was well ahead of its time. The design of the communication system was determined largely by the needs of fine grain systolic computing, and was not committed to message passing nor to coherent shared memory. This made it an extremely interesting hardware platform to develop, test and experiment with new models for communication in parallel and distributed computing. It had excellent support for a broad range of communication primitives, that were all put into the design without stating a precise specification what it should be good for. I experienced this as an intellectual challenge and found it highly interesting to take some of these capabilities and put them together into new solutions for more effective communication in parallel and distributed application programs.

1.2 Problem Statement

Assuming that most code for parallel and distributed systems will be written by some parallelizing compiler or a high level parallel program generator tool chain the actual machine model (coherent shared memory or message passing distributed memory) will become less significant to the application writer. The focus will shift to providing the best possible data transfer services with the least complex and least expensive hardware technology. This thesis deals with the search for the best possible transfer services for parallel program generators and comprises several models to establish a better conceptional structure in the problem space of possible solution. As experimental study it also describes several experimental communications system on five (!) different parallel or distributed computing platforms, that serve as data points to verify the proper handling of alternatives and tradeoffs.

In this (slightly unusual) revision of a Doctoral thesis about 10 years after my initial work started on that topic I do concede that High Performance Fortran (HPF) and the well promised ubiquitous programming generators, that were to make parallel programming as easy as scripting a few GUI gadgets into an Internet browser or as easy as putting together a simple client server application did not really materialize. Instead a larger number of programmers coming from other disciplines (mainly natural sciences) have been trained and hired to program and maintain high performance computing codes, like message passing parallel programs. For the validity and the significance of the arguments in this thesis this trend is no problem. Ultimately the programming of parallel and distributed applications by humans or the programming through tools like program generators will raise exactly the same problems. Many rules and guidelines of this thesis can therfore be used by human application programmer as well as a compiler back end. The principles that linke the programming model to the implementation performance will ultimately help to write more efficient parallel application codes. A systematic view of alternatives, tradeoffs and cost-benefit models regarding communication hardware is further needed by the message passing system developers, that try to write more efficient system software for emerging platforms like clusters of PCs and other future platform for distributed or collaborative computing.

1.3 Thesis Statement

The winner for the most efficient, i.e. the best data transfer services with the least amount of hardware support, is neither a pure coherent shared memory architecture nor a pure, coarse grain message passing distributed memory architecture. Looking at end-to-end transfers, the optimum lies in between the two extremes. Fine grain data transfer mechanism that rely on non coherent remote loads and stores in a global address space are highly useful mechanism. New models of communication that separate control and data transfers are required to link the property of those data transfer mechanism to the property of parallel programs and their correctness. My deposit and fetch model will successfully to do this. The evaluation of several implementations of direct deposit indicate that direct deposit results in a major win (factor of three on a Cray T3D) for large data transfers with complex communication- or memory access patterns. and that the benefit is largely due to a reduction of data copies in the internals of the communication system.

The search for the optimal performance in message passing systems can be approached from two ends. First the performance of a full function messaging library can be analized and the costly operations can be carefully eliminiated. Second an implementor can start from the most efficient low level primitives and add functionality until a reasonnable programming model is offered. Personally I have worked from both ends and arrived in both cases at direct deposit message passing.

1.3.1 Reducing full function libraries to direct deposit

The first approach of finding a faster communication for message passing systems is to analyze the sub-optimal performance of existing message passing systems, as done e.g. by the authors of [63]. The study resulted in the development of Urbana Fast Messages [84] that successfully avoided some overheads of remote buffer allocation. The advantage of this development methodology with the preceding in depth analysis is that there will be a detailed understanding of performance and maybe even an allocation of the cost to certain functionality. Still reducing a messaging library to its basic functionality may not always lead to the best possible performance because the implementation usually remains withing the general model. The reduction approach makes it easy to preserve some defined semantic properties of a message passing system at the expense of performance gain.

1.3.2 Constructing a programming model for remote stores

The second approach is starting from fast primitives, like e.g. the remote store hardware support that is found in the T3D and adds supplemental functionality plus a programming model until a suitable programming environment is obtained. My implementation approaches on iWarp and on the T3D certainly went that route. Another attempt for a bottom up programming model for the T3D was the effort to establish F– by B. Numrich [81]. The advantage of the constructive approach is that overheads and performance remains under strict control of the implementer. Starting at the best possible performance figures for the basic data transfers, the development goes hand in hand with an modeling effort that can carefully spend overheads on additional services until some better semantic properties, and a clean model is reached.

1.4 Related work

Computer Architecture

A new messaging model that takes the best of two highly popular worlds is related to most recent work in parallel architectures. In the mean time several advanced computer architecture textbooks are available for further reference (e.g. [25] or [51]. I am limiting this abbreviated survey of parallel computer achitecture to a brief contrast of my compiler directed message passing approach with two alternative approaches of using coherent shared memory in hardware and the alternative of a software DSM system.

The Stanford FLASH architecture prototype [69] provides cache-line based, fine grain shared memory and moves the data through a cache-coherency protocol upon accesses to the shared address space. For improved performance fully cache coherent approach is relaxed and explicit communication primitives are added through specialized custom coherency protocol [49, 50]. The FLASH project leads a trend away from automatic coherency in shared memory prototypes. On the other side newer commercial private memory machines such as the Cray T3D or T3E tend to provide drastically improved architectural support for coarse grain shared address space and remote memory operations. The two classes of parallel machines seem to be converging.

The shared address space of the Princeton SHRIMP architecture is of coarser granularity than FLASH. The basic unit of shared objects is a virtual memory page (more precisely: the object can be smaller than a page, but only one object per page can be shared) rather than a cache line [13]. Similar to the capabilities offered by MAGIC used for custom coherency protocols in FLASH the SHRIMP network interface allows updates to be automatic or under program control. The different choices in consistency models can be used to improve communication efficiency in applications [59]. An important goal for the SHRIMP architecture is compatibility with a standard PC environment and a workstation operating system [33].

On a first look my direct deposit message passing architecture might look like a form of release consistent shared memory. The major difference is that we closely integrate the a minimal hardware support with the back end of a compiler to provide a shared logical address space only to the HPF programmer but not to the intermediate code. The resulting machine model and is capable of sharing objects at an extremely fine granularity without without coalescing them to intermediate data structures. Internally we preserve the notion of a message between two processors and permit the compiler to aggregate the transfers, amortize any startup overheads and optimize memory systems performance during the copy loops. Therefore the direct deposit system as a few more hooks for software controlled optimizations than the hardware only or operating system only implementations of shared memory.

Related operating system work

The performance problems of tightly coupled synchronization and data transfers have been independently recognized by other researchers [104]. Their investigations targets distributed operating systems and transaction processing rather than data parallel computations. The predominant method of communication in these applications is RPC and not message passing. When introduced, the coupled notion of control and data transfers RPC were considered the major improvements over simpler mechanisms like sockets [79]. At this time the blocking semantics of RPC and the limiting factor of coupled control and data transfer seem to make the original form of RPC unattractive for high performance distributed computing. Special RPC style programming is also found in early inefficient signal based messaging primitives [38] and the much better optimized active messages [108]. Systems dispatching computational tasks using fine grain work scheduling (e.g. CILK [12]) link control and data transfers just like RPC mechanism.

A variety of models for coherent shared memory were brought to message passing distributed memory system by software DSM systems. Unlike true cache line based coherent shared memory machines those systems do not need much hardware support but work on NOWs (Networks of Workstations) and CoPs (Clusters of PCs). Those system also fall into the design space between coherent shared memory and message passing since most of them implement a consistency model that is much weaker than in shared memory machines with full hardware support. The system are an interesting experimental target but as they are built on top of conventional messaging mechanism their hybrid models can not surpass the performance of message passing systems. The most prominent representative of software DSM systems are Ivy [72] (that started it all), Treadmarks [3] and Midway [10]. It is left as interesting question for further research if message passing infrastructure can be specialized for these software DSM, but it is conjectured that the resulting communication infrastructure could be faster than a conventional message passing system. Such would confirm my thesis that the optimal communication support is neither pure shared memory nor full function message passing.

An interesting study about the set of transformation that carry a shared memory program for a DSM released consistency to a conventional message passing program by adding annotations until send- and receive primitives become visible is given by H. Karl in [64]. The paper implements the pure DSM code, the pure message passing code and a few intermediate forms on the Charlotte DSM system [8] and gives performance figures some small benchmarks, quantifying the performance tradeoffs between a shared memory model with lots of coherence and a bare bone message passing code without coherence. However this newer study related to Web based meta-computing using Java, a VM and PCs interconnected by an Ethernet. No absolute performance numbers are given.

Memory based communication - a tutorial

There exist two major approaches to the design of a communication system. Consider a parallel system with n *nodes* and some memory *M*. In all realistic systems with more than a small number of nodes, the memory is distributed, i.e. the time to access a memory location is no longer uniform. To achieve good performance, it is desirable to keep a data item *D* "close" to the node *N* that operates on this item. If *D* is in a memory location M_D that is expensive to access from *N*, then it is advantageous to move *D* to a different location.

The two approaches differ in the level of support given to such movement of data: as an item *I* (e.g., a variable or part of a variable like an element of an array) is moved to a location close to node *n*, *I* either keeps its original name (address), or it is renamed. The first approach, commonly known as *shared memory*, uses, for example, a cache *C* for each node. As an item *I* is moved from its home location M_I in *M* to a node's cache, the address A_I of I remains unchanged. That is, the node continues to refer to this item using address A_I. The second approach requires that an item is explicitly renamed. That is, when the item *I* is moved into the local memory of a node *N*, I is given a new, distinct name (i.e., address) M_{new_I}, and programs executing on *N* refer to *I* using this address M_{new_I}. This approach is usually referred to as *message passing*, in reference to the primitives used to transfer data from one node to another. Figure 1.1 depicts these two scenarios.

In the shared memory approach, the communication system maintains consistency. That is, any reference to an item *I* using its address A_I yields the most recent version of this item. In the message passing system, there exist multiple copies of an item, and the program has to distinguish between these copies by using the address of a specific copy. Message passing has the advantage that the communication system can concentrate on the exchange of data; managing the consistency of the data space is the task of another part of the system. Of course, the requirement to explicitly manage consistency complicates the task of programming the system.

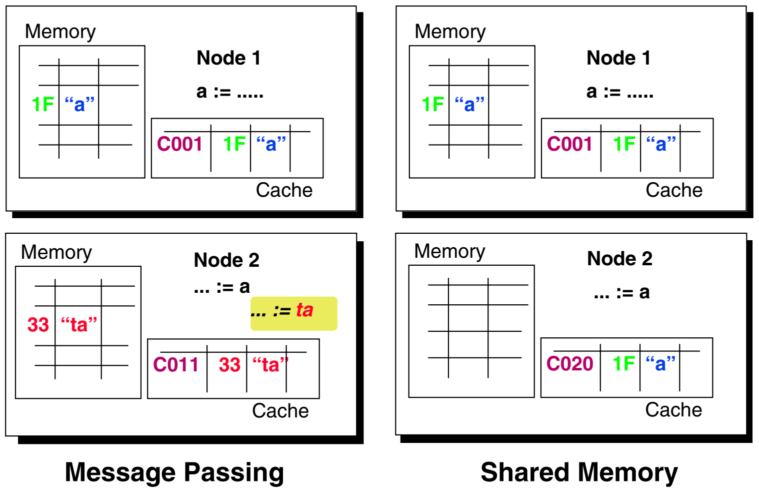


Figure 1.1: Two approaches to improve locality of reference: coherent shared memory and message passing distributed memory.

Therefore, we differentiate between message passing and shared memory by whether data items are renamed as they are moved from one node to another. This view abstracts from the policies on when to perform such a transfer as well as from the mechanisms that perform the transfer. The policy can be under control of the user (e.g., when the user program transfers explicitly), managed by a compiler or runtime or operating system, or taken care off by the memory hierarchy (e.g., the cache consistency protocol). Of course, combinations of these and other options exist as well, and a number of algorithms to address this problem have been described in the literature.

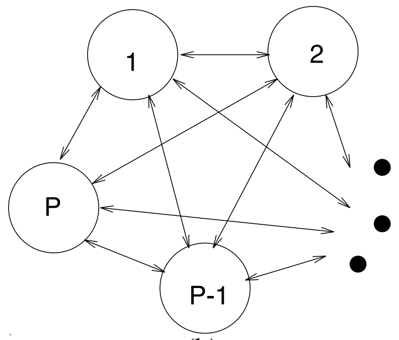
There also exist a variety of mechanisms to implement such transfers. Many variations of the conventional message passing operations *send* and *receive* are well-known and documented. Alternatively, a system may implement message passing by memory copy operations: a node N_1 maps some locations $L_2^1, L_2^2, L_2^3, \ldots L_2^n$ of the local memory of another node N_2 into its own address space. These locations can now be accessed by N_1 as $L_1^{k+1}, L_1^{k+2}, L_1^{k+3}, \ldots L_1^{k+n}$. To transfer an item *I*, residing in L_1^i on node N_1, *I* is copied from L_1^i to a location mapped into N_1's address space (say L_1^{k+1}), and the communication system transfers the value to the memory of node N_2. The important aspect is that there are now two copies of *I*: one accessible only to N_1, in L_1^i, and another copy that can be accessed as L_2^{k+1} by node N_1 and as L_2^1 by node N_2. Notice that the program used memory (copy) operations to effect the transfer of the data. Another mechanism that has been discussed is the use of *prefetching* to improve the performance of memory accesses. Such pre-fetch instructions are inserted into the code into a location sufficiently far ahead of the expected use of an item. If everything works as planned, the item is already in the cache when the item is referenced by the program. While pre-fetch instructions are explicit, they do not cause a renaming of item, so they fall in the category of shared-memory mechanisms.

Neither message passing nor shared memory solve all the problems of parallel programming. A discussion of the relative advantages of shared memory and message passing continues to take place, and there exist a number of papers debating this topic. As long as compiler technology continues to improve, this discussion will not be settled, since the relative merits of each approach depend on to what extent it can serve as a target for programming tools. There are also proposals to combine these two architecture styles in a single framework.

In any case, data transfer is a key issue for the communication system. In this thesis, we address

```
parallel do i = 1, n
   fft(A(:,i))
enddo
transpose(A,B)
parallel do i = 1, n
   fft(B(:,i))
enddo
transpose(B,A)
```

(a)



(b)

Figure 1.2: Basic operations of the 2D FFT program.

techniques to improve the performance of such transfers; we focus on the core of message passing and leave the discussion of consistency management to a later time.

Compiler-generated communication - a tutorial and survey

My motivation for looking at communication in parallel systems comes from developing the Fx data parallel compiler [101]. Fx compiles a variant of High Performance Fortran [41]. The issues of compiler-generated communication are slightly different from explicit message passing communication.

In the data parallel model, the programmer writes a single-threaded program, and the compiler is responsible for dividing the computation over the available processors and generating the necessary communication calls. From the programmer's point of view, the resulting communication is implicit, a side effect of array assignment statements. The data parallel programming model eases the task of communication analysis for a compiler by providing a global name space. The data parallel compiler translates each data parallel statement into a communication step and a computation step, so the compiler views the program as a series of alternating communication and computation steps. With the proper analysis tools [17, 16, 66, 65, 112] or user directives[41], a data parallel compiler can discover exactly what communication must take place in each step and describe these communication requirements as sequences of *communication patterns*, sets of node-to-node communication exchanges.

For an example of compiler-derived communication patterns, consider a two dimensional fast Fourier transform (2D FFT). Figure 1.2(a) outlines the data parallel code for this program. First the program performs 1D FFTs over the columns. Then it performs a transpose and calculates the 1D FFTs over the new columns. Finally, it performs another transpose to return the data to its original array. If the columns of arrays A and B are distributed over the processor array, the parallel loops can be executed in parallel, and the transpose operation requires an all-to-all communication pattern shown in Figure 1.2(b).

Since the compiler has global information about the communication patterns, it can perform optimizations that cannot be performed in a runtime library. Also, once the compiler has been targeted to a particular system, it is better able than a human programmer to avoid error conditions. The compiler-generated code does not need all the error and safety checks of a general message passing library, and the communication interface only needs to be consistent but not necessarily user-friendly for compiler-generated communication.

However, some aspects of compiler-generated communication introduce more problems than explicitly written message passing communication. I cannot expect the compiler to alter an algorithm to avoid "bad communication cases" as a human programmer might. Most knowledgeable message passing programmers will adjust the parallel algorithm to ensure that only large, contiguous blocks of data are exchanged. While a human programmer can consciously avoid bad communication cases on a particular machine, the message passing communication must be re-optimized for each new machine, since communication performance varies dramatically between different parallel systems. Once a compiler is re-targeted and tuned to a new machine, all communication code generated by the compiler will be automatically re-optimized for the new machine.

In data parallel programs, communication of blocks of non-contiguous data occurs when arrays are distributed across the machine in different ways. Therefore, one of my goals is to determine how to provide high communication performance for non-contiguous blocks of data.

The remainder of this thesis is structured as follows. In Chapter 2 I give an overview of the parallel machines used in this thesis. This first discussion includes the node and the communication architecture. In Chapter 3 I describe the anatomy of a fully functional message passing system. In Chapter 4 I consider some common models of message passing and derive the *deposit* and *fetch* model by relating the properties of the transfers (implementation properties) to the properties of the programs that can execute correctly. I also establish a reference models for the steps involved in message passing. In Chapter 5 I argue that about the high potential of performance improvements that can be achieved by decoupling control and data transfer. In the Chapter 6 I define the common data access pattern for fine grain data accesses that typically occur in compiled parallel programs and introduce the copy-transfer model to describe the performance of all local and remote memory-to-memory transfers in uniform terms. In the Chapter 7 I describe the best implementation of direct deposit on various machines. In the Chapter 8 I give the details about the maximal performance of the memory system hardware of the parallel machines under consideration, the iWarp, the Cray T3D, the Cray T3E, the Intel Paragon and the DEC 8400 and finally in Chapter 9 I can properly relate the deposit mechanisms to the actual hardware instrumentation and to the performance that should be possible. I also evaluate the performance of direct deposit messaging for the best memory-to-memory transfers based transpose and other application kernels.

Chapter 2

Parallel Systems used for our Evaluation

Message passing and shared memory models are the most common application programming interfaces (API) in mainstream parallel computing. Many programs are written for a parallel platforms because they require a large amount of computation and part of that computation can be done in parallel. In most cases the demand for processing power and the amount of memory exceeds the current standards for single processor workstations and PCs or the turnaround time of an engineering cycle must be lowered below acceptable values. In both cases parallel systems must provide a large amount of compute power as cheaply as possible. The latter goal of high performance for a low price is achieved by general purpose microprocessors. The high volume of those chips makes them much cheaper than dedicated special purpose architectures. Therefore all parallel systems considered in this thesis are built around a general purpose microprocessor. The chapter contains a brief overview over the high level architecture of the different microprocessor based compute nodes. Chapter 6 will refine this high level view with a quantitative discussion of their memory system in more detail.

With the exception of a few embarrassingly parallel programs most production codes of parallel computing require large amounts of data to be transferred during the course of a parallel computation. The efficiency of those application critically depends on fast network interconnects to move such data at high speeds. The parallel and distributed systems under consideration are all built around a particular fast interconnect technology. Although the contribution of this thesis is nearly independent of network topology and technology an overview is given for completeness.

This discussion of the hardware sketches the layout of the main functional units in a Cray T3E, a Cray T3D, an Intel Pargon, an Intel iWarp and a DEC 8400 parallel computer. For a more detailed description consult the references given after the sketch. In this general view *no distinction* is made between network computers and symmetric multiprocessors based on the amount of coherency supported in hardware. In the case of a bus based symmetric multiprocessor the bus is just a network, the caches are local memory and the memory boards are the remote memory. Classifying the machines according to the different layouts in Figure 2.1 the Cray T3D, T3E, the Intel iWarp and the Intel Paragon are all network computers, while the DEC8400 is a symmetric shared memory multiprocessor.

2.1 Microprocessors, memory system and network interface

Looking at the performance numbers in the subsequent chapters, there is no doubt, that the most suitable memory systems to justify “direct deposit” are found in the Cray T3D and T3E machines, although the most important ideas and models were developed earlier on the iWarp research prototypes. I consider the T3D and the T3E to be an evolution of processor, the memory system and network interfaces. Such a

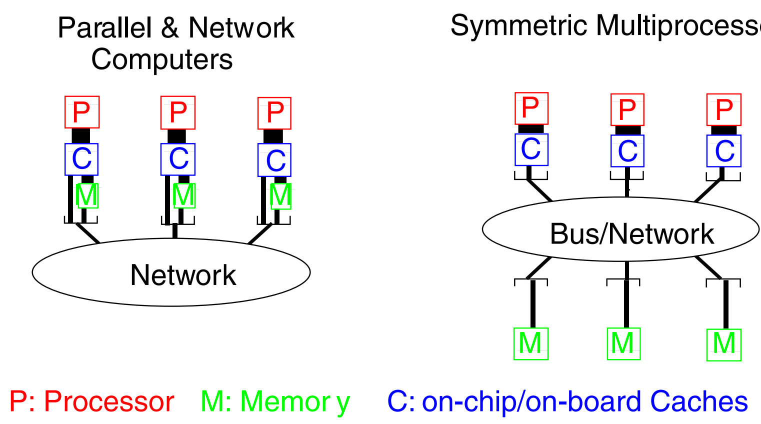

Parallel & Network Computers
Symmetric Multiprocessors
P
C
M
Network
Bus/Network
P: Processor
M: Memory
C: on-chip/on-board Caches

Figure 2.1: The overall layout of the memory system and the amount of hardware support for coherency are the factors that distinguish between distributed and shared memory architectures. The memory system performance can vary in both types of machines.

description permits us to keep it short. A complete description of the hardware belongs to the hardware reference manuals of the vendor and is far beyond the scope of a thesis in systems architecture.

The Cray T3D

A T3D node comprises a 150MHz 64bit DEC Alpha EV-4 microprocessor (21064), a local memory system, a memory mapped network interface to send remote stores to the network, and a deposit engine. The latter two are usually referred to as *the annex* in the construction plans of the vendor and other in many other publications. The memory of a T3D node is a simple memory system built from DRAM chips without extensive support for interleaving and pipelined accesses. Unlike workstations, the node has no virtual memory and runs with a special version of the DEC Microprocessor without the functional units for paged virtual memory.

The interface between the computation agent and the main memory is an 8KB primary cache, which is implemented on-chip within the DEC Alpha microprocessor. The memory system and its interface to processor and communication are shown in Figure 2.2. An external read-ahead circuitry (RDAL) can be turned on by the programmer at load-time to improve performance of contiguous load streams; I measured improvements of approx. 60%. For writes, the default configuration of the cache is write-around, and support for writes consists of the write back queue (WBQ) provided by the microprocessor. The documentation of the Cray T3D Application Programmers Course [23] specifies the local read bandwidth at 55 MB/s for non-contiguous single word transfers, and up to 320 MB/s for contiguous reading of cache lines with read-ahead. The latency of a load from main memory is around 150ns.

The interface between the processor and communication system of a Cray T3D node was designed as the so called annex - a memory mapped communication port, the maps some range of free address space to the physical memory of another node in the system. In most cases one other node is selected as a communication partner. Once a store operation is issued to that communication port, the communication subsystem takes over the specified address and data, and it sends a message out to the receiver. Remote loads are handled in a similar way and can be pipelined with an external, 16 element FIFO queue. This fetch queue requires explicit programmer or compiler support.

The memory mapped communication port automatically deals with the outgoing address and data

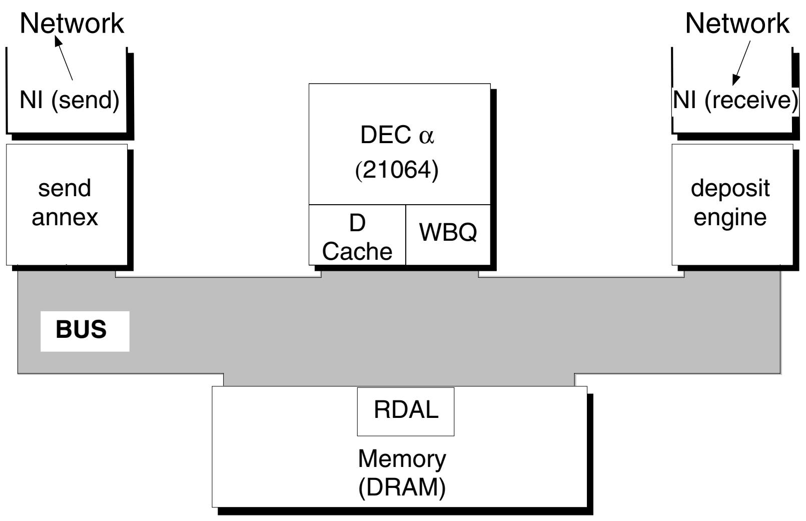


Figure 2.2: Overview of the Cray T3D node architecture: *NI* refers to the network interface circuitry and with control logic and FIFOs for data transfers.

words. As a port it maps some range of free address space to the physical memory of another node that is selected as a communication partner. The communication partner can be switched with a fixed overhead by modifying the appropriate annex entry. The significant fixed cost for switching the communication partner and the lack of global cache coherency justifies my classification of the T3D as a highly advanced distributed memory, message passing machine.

For remote stores, the deposit engine on the receiving node completes the operation on behalf of the user. Such remote accesses happen without any involvement of the processor at the receiver node, i.e. there is no requirement to generate an interrupt to the processor. The deposit circuitry can store incoming data words directly into the user space of the processing element, since both, the address and data word are sent across the network. The direct mapped on-chip cache of the main processor can be invalidated conservatively on a line-by-line basis as data are stored into local memory, or it can be invalidated entirely when the program reaches a synchronization point. The user program must control the invalidation strategy.

Transfers from the processor to the communication system can be performed at a rate of approximately 125 MB/s, and if multiple nodes perform remote stores of contiguous blocks to a single node, these transfers can even be processed at the full network speed (160 MB/s)[80].

Coherency between local and global memory accesses must be reached explicitly by global barriers. The barriers are supported in a special purpose network and are blazingly fast since they are entirely implemented in hardware. The special purpose synchronization network is partition-able and provides sufficient subsets (barrier bits) to adapt to most runtime partitions of a production system. The entire machine can be synchronized in about two remote store or about two message latencies. The combination of a global barrier with counters for outstanding references provides an excellent tool to achieve global coherency at global synchronization points. However the local coherency between cache and memory must be achieved separately by the cache invalidation mechanisms listed above. An architecture with such a large amount of programmer or compiler action to maintain coherency can hardly be called a shared memory machine but remains a message passing machine with exceptional hardware support for low latency and fine granularity.

References:[29, 82, 1, 9]

The Cray T3E

A T3E node comprises an advanced 64bit DEC EV-5 Alpha microprocessor (21164), a local memory system, a memory mapped e-register interface to dispatch remote loads/stores to the network. Just like the T3D it also includes a deposit/fetch engine to handle those remote memory operations directly without involving the memory system of the communication target. The memory of a T3E node is an moderately interleaved memory system built from common DRAM chips. The node architecture has support for virtual memory including an elaborate local to global virtual address translation, but unlike workstations, the microprocessor of a node is usually not configured for on demand paging. The technical specifications show capabilities for the translation of global to local virtual addresses that permit the implementation of distributed arrays in hardware. The separation of different address bits resembles the process of centrifuge pushing certain selected bits inwards (towards LSB) and certain selected bits outwards (towards USB) and therfore this hardware is called the *address centrifuge* in the literature about the T3E. An early version of this logic was already present in the block transfer engine of the predecessor, the Cray T3D, but that DMA like block transfer engine was accessed in privileged mode only and therfore remained impractical and unused by most application. The T3E fixed this problem. Still in data parallel Fortran languages, like e.g. Fortran Fx, the data redistributions are normally tied to array assignments and achieved by copying rather than by mapping - therefore our existing Fx Fortran toolchain does neither require nor support the address centrifuge.

The interface between the computation agent and the main memory is an 8KB primary cache and a 96KB secondary cache, both implemented on-chip within the DEC Alpha EV-5 microprocessor. The memory system and its interface to processor and communication are shown in Figure 2.2. The E-registers can also be used as a read-ahead circuitry to improve performance of contiguous load streams. Such load streams can bypass the caches for maximum efficiency.

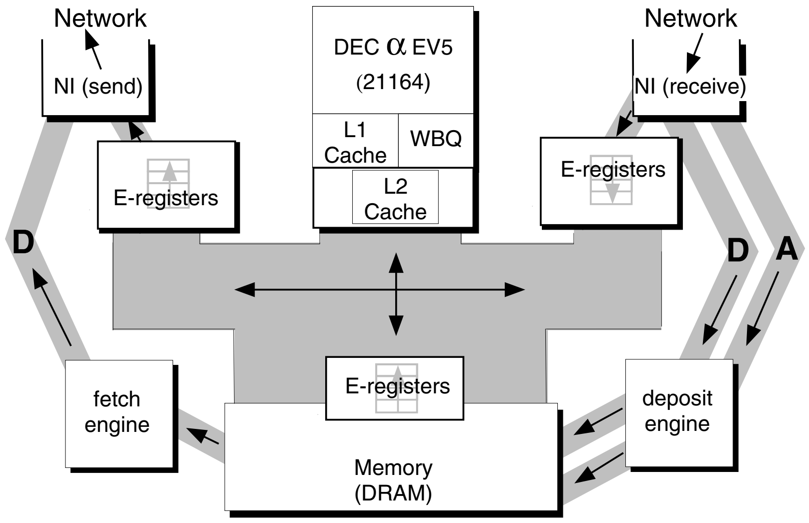


Figure 2.3: Overview of the Cray T3E node architecture: *EREG* refers to a block of fast registers in the support circuitry outside the microprocessor. *NI* refers to the network interface circuitry and with control logic and FIFOs for data transfers..

The newer EV-5 Alpha processor has a larger number of address pins to represent physical addresses in a system. Still, despite its 64bit internal addressing, the number of address pins is not enough to address the entire global memory of a potentially large massively parallel machine. Instead of the the partial mapping through the annex, the Cray T3E must ship global memory addresses through the 64bit data path to the E-registers in the support circuitry. Remote memory operations are no longer transparent to the assembly programmer, but due to the high clock rate the required multi-instruction sequences are competitive, when it comes to performance. It is self understood that the compilers can generate the code sequences for remote stores and loads easily. Due to the big latency of several hundreds cycles for loads to remote memory compiler techniques like prefetching or pipelining are required to achieve full bandwidth. So the only thing lost with the newly introduced E-Registers was the amazing simplicity of full performance remote stores in its predecessor. Just like in the T3D the communication partner can be switched with a fixed overhead by modifying the e-registers in control space. At the conclusion of my experimental work the precise overheads were not known to normal users of the machine.

Similar to the T3D, every T3E node has some fetch/deposit circuitry that handles incoming remote operations (loads and stores) with their memory accesses on behalf of the communication system. Again these accesses can happen without involvement of the processor at the receiving node, i.e. there is no requirement to generate an interrupt. The deposit/fetch circuitry can store incoming data words directly into the user space of the processing element, since both address and data are sent over the network.

Coherency between local and global memory accesses is still reached explicitly by global barriers. However the special purpose network with its partitionable, tree like FAN IN/OUT circuitry is no longer supported. A new implementation of synchronization operations over the regular high speed interconnects delivers almost the same absolute performance and much more flexibility (e.g. for the mapping of redundant nodes). The combining operations of the barriers were moved from the network to the network interface, but they are still done in hardware. The underlying mechanisms also provide low latency messages for signaling. According to the hardware specifications such messaging is available in user space for normal programming, but since I never received the necessary low level libraries I could not use it in my runs with an early production machine.

In retrospective I would classify the Cray T3E as the most advanced fine grain message passing machine ever built and not as a distributed shared memory machines due to the incomplete support for global coherency.

References:[90]

The Intel iWarp

Among all machines considered in this thesis, the Intel iWarp is the oldest system. First prototypes became operational in 1990, while the production systems were delivered during the summer of 1991. Its 32bit microprocessor core is running at 20 MHz, a clock rate that is clearly inferior to some other systems described here, but this advantage is compensated by its unique design with an integrated network interface and a design purely based on fast static SRAM memories that made iWarp systems faster than many of its successors. I can give a most basic introduction into the hardware support involved in message passing and must refer the interested reader to a more complete book about the iWarp project [47]. The memory system and its interface to processor and communication are depicted in Figure 2.4.

There is no need to say much about the simple 32bit RISC processor core. Its data path is capable of executing one instruction per cycle at 20 MHz. At this low clock rate, the latency and true execution time of most basic operations could be kept at a single clock as long as operands simply go back to the register file or are bypassed between immediately following instructions. The performance of the basic processor core is comparable with the first implementations of Sun SPARC or the early MIPS processors. Both types of processors had life cycles overlapping with the live cycle of the iWarp component. Besides its regular 32bit RISC like instruction set, the iWarp processor provided a 96bit LIW instruction with address

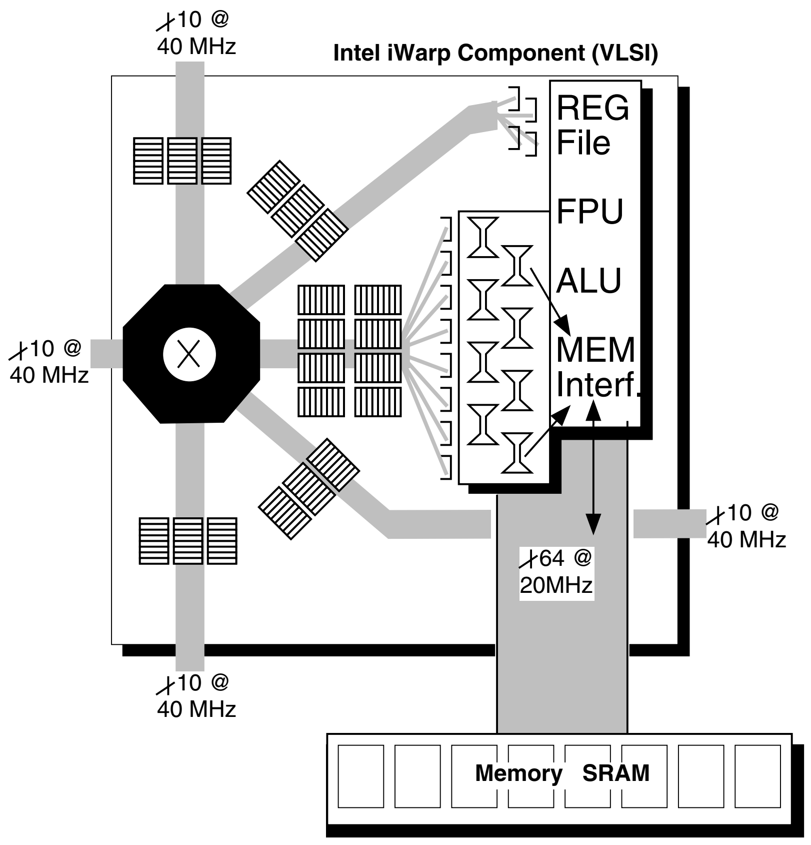


Figure 2.4: Overview of the Intel iWarp node architecture: The entire network interface is integrated into one single VLSI component.

generation, communication, loop processing overlapped with two floating point operations in some tight inner loops. The LIW capability of this processor is primarily designed for most efficient computation loops (or to be precise, for a combination of communication stream and computation as they occur in many fine grain systolic algorithms). The LIW instructions support the message passing software in just one way. Software pipelines LIW instructions with memory accesses can handle data transfer at peak speeds, even in the presence of most complex data access patterns with fixed, computed or indexed strides. Memory access latency to the entire main memory is 2 cycles per 32bit word or 3 cycles per 64bit word. The data path between processors and memory is 64bit wide and has a bandwidth 160 MByte/s. Computation alone could only reach two thirds of this impressive memory bandwidth, but the remaining third remains available to memory based communication through some DMA like agents, called *spools*. There are eight spools integrated into the processor core. In relative numbers (clocks) this is about the speed of L1 cache on other machines. In absolute numbers the 100ns latency and 160 Mbyte/s bandwidth of main memory of iWarp remains comparable to speed of L2 and L3 caches of many newer systems considered in this thesis.

The iWarp microprocessor integrates a network interface and a router with the processor core in one single VLSI chip. The external connectivity of the chip is through its 160 Mbyte/s memory interface and four bidirectional communication channels operating at 40 MB/s in each direction. In all, but a very few combination of routes through a single node, the on-chip switching fabric is able to forward the flits of all eight communication channels at full speed, with a total switching capacity of 320 MB/s in each node. In theory the network interface to the processing element can contribute traffic to the switch by sourceing or sinking up to four 40 MB/s streams. In practice sourceing and sinking two streams at full speed is realistic and sustainable in real programs. A description of the interconnects will introduce the logical channels, flit forwarding and how they influenced the routing strategy messages passing. Routing for streams is discussed in earlier work of the author [97] and in the final report on the iWarp project [47].

As a research prototype, the iWarp component incorporated many new ideas into its network interface. The rich set of primitives poses a challenge to the programmer of low level communication software. Most of the technological advances in the network interface are architecturally justified through its on-chip integration with the processor core. With all circuitry for a complete node of parallel computer on a single chip, there were no pin-count or signaling-speed issues limiting the design of the iWarp network interface. A few hundred bits of communication state are accessible as a block of control space registers by some simple register move instructions taking either one or three processor cycles. Communication state can be polled entirely within the processor and without using any bandwidth of the memory system interface. At least in theory, the interrupt latencies could be kept extremely low, as long as interrupt dispatch is optimized. An on-chip ROM with direct path to the instruction dispatcher could contain direct deposit or active message communication handler and make them available with no task switching cost, even without I-cache misses. Furthermore large register file with 128 registers would have provided dedicated registers. Unfortunately this unique opportunity had to be given away, since nobody (including the myself, as author of this thesis) was able to commit to final communication and message passing code, when design of the CPU had to be frozen. Therfore the high speed on-chip ROM was used for CPU diagnostics routines instead of communication handlers.

The network interface is divided into two complexes of functional units. Both groups of functional units are useful for various communication styles, regardless whether they are connection-oriented or messaging-oriented. The first group deals with programmed communication from and to the processing core of the microprocessor. The second group deals with background communication from and to memory. Both groups of functional units access the communication system through an inbound or outbound FIFO buffer, referred to as a Pathway Control Table Entry (PCT) by a naming after the state table controlling those buffers. Theses FIFO buffers can hold 8 words and can be checked for full or empty.

The first group of functional unit for programmed communication comprises two pairs of systolic gate registers that map any logical communication channels directly into the register file of the microprocessor. Like in the transputers [58] it is possible to use a data word coming from a communication channel as an operand or make a communication channel the target for the result of any processor operation. Synchronization between the instruction stream and the communication system is achieved through an interlock that stalls accesses to those gate register if no data or no space is available. For deadlock detection and removal, the iWarp hardware would provide time-outs to bail out of an unwanted stall, but those mechanisms were never thoroughly tested and therfore left unused.

With a set of special move instructions it is also possible to transfer specially tagged data words to the communication system. Such tags will give those headers and trailers a special meaning and permit to send commands to the router for delimiting messages and for selecting routes. Like regular data words, tagged data words can be send with a simple instruction at full speed. Therefore the use of message headers and trailers remains highly flexible on iWarp and is performed entirely in software, whereas most other systems rely on fixed hardware solutions incorporating control registers to select the destination and lookup in hardware routing table.

With programmed communication the possibility to check the FIFO for full and empty becomes highly important. Such checks give the programmer a guarantee that a certain number of words (e.g. a whole header) can be read or written without stalls. Similar to other Intel system like e.g. the Paragon those checks are often incomplete and sometimes inaccurate. Because of interlocks and stalls a deficient check of the FIFO status would not directly result in corrupted data, but in a message passing environment such could easily imply a routing deadlock.

The second group of functional units in the network interface comprises eight spools. Spools are similar to DMA agents in other systems and move data asynchronously from memory to the communication channels. The spools run asynchronously to the instruction stream, perform their own address computations and eventually flag their completion through status bits to the processor core. The spool status can be polled or can raise exceptions or interrupts. Spools operate between the memory and the communication system. Since the communication system is fully integrated with the microprocessor the data for spools is fetched through the memory system interface of the CPU processor core. Instead of a costly arbitration between spools and the processor core for memory access the iWarp spools are implemented as special load instructions that are automatically and asynchronously inserted into the instruction stream of the microprocessor. For the software the eight spools can be modeled as independent asynchronous threads moving data from memory to the communication system or vice versa. With this trick the memory bus is automatically arbitrated and the integer units of the microprocessor can be used for address generation within the spools. The loss of a machine cycle for every transfer of a double word between memory and communication system is not a severe performance disadvantage. Since there are no caches, most of the common computation loops take memory accesses to the limit and therefore additional memory accesses of the communication system would cause stall cycles in any event.

While designed for systolic computation, the iWarp system has a remarkable network interface that makes it one of the best and most innovative message passing machines every built. However as a research prototype the iWarp node design relied on two unrealistic assumptions, that limited it commercial viability. It relied on SRAM for memory and on a non-standard processor core that could not keep up with the fast pace of commodity microprocessors.

References:[45, 15, 47]

The Intel Paragon

The node of a Paragon system contains multiple off-the-shelf microprocessor processors sharing one common memory system. My investigation is based on a system with two processors per node, but systems with three processors per node have been built as well. Except for the mechanisms to support multiple processors, the memory system of the Paragon system is surprisingly similar to the one in the Cray T3D. The memory system and its interface to processor and communication are depicted in Figure 2.5.

The processors of our Paragon nodes are two Intel i860XP processors. Both processors have their own primary on-chip data cache and are connected to the local memory system over a 400 MB/s high speed bus. The data cache is 16 KB, organized 4-way associative, write-back or write-through. Under SUNMOS [74] (the operating system of choice for low-latency communication) the caches are write through. The i860XP processors contain support for higher bandwidth through pipelined loads (using the PFQ) that bypass the caches.

The interface between the processors and the communication system is realized by memory mapped ports, which are mapped to the FIFOs of the network interface. A remote store can be performed from the processor to the communication system through the main high speed bus.

The memory system contains two DMA controllers (also known as line transfer unit), which can serve as deposit engines (with some restrictions). The two DMA controllers can handle both in-coming and outgoing transfers, but are not as powerful or as flexible as the annex circuitry of the T3D or the spools of an iWarp component. They require a processor for setting up a transfer and also for handling page boundaries and numerous exceptions, which is a quite expensive solution. Most importantly the Paragon DMAs can

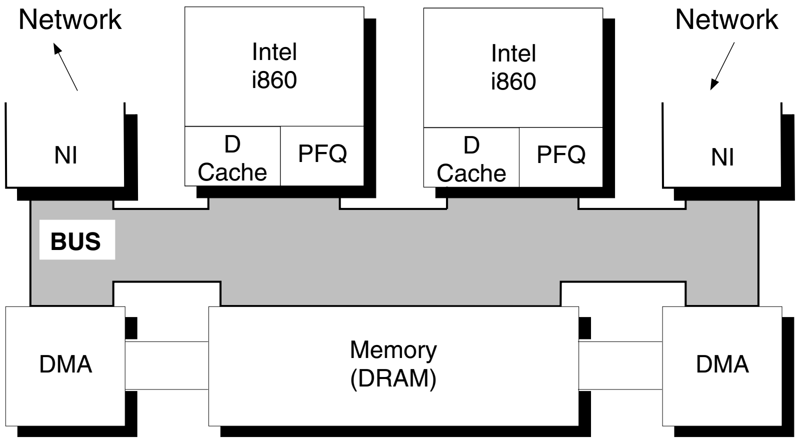


Figure 2.5: Overview of the Intel Paragon node architecture: *NI* refers to the network interface circuitry and with control logic and FIFOs for data transfers.

handle only well aligned, contiguous block-transfers.

Those features make an Intel Paragon a representative of the classic distributed memory system that can only be used message passing programs. Such systems will later evolve into Network of Workstation and Cluster of Commodity PC architectures.

References: [20]

The DEC 8400

In contrast to the four systems listed above, the DEC 8400 was designed as a high performance workstation with a symmetric multiprocessor architecture. There are some differences and some similarities to the massively parallel distributed memory systems. A DEC 8400 does not contain a network interface, since all communication between processors must be arranged by data copies through the shared memory banks in the common memory system. The architecture is designed to run multi-threaded programs with a single address space through coherent shared memory. Implementing a message passing system and communicating by explicit copies is an alternative way to program such a system. From the hardware point of view this is not necessarily the best way to use such a system, but nevertheless it is a common way to run applications that are coded in a machine independent way using a standard message passing API.

Through its broadcast bus the DEC 8400 gains an easy hardware solution for coherence in shared memory, but in turn it looses its scalability and is limited to systems with about 12 processors. The compact nine slot bus design permits some flexibility to trade off more memory banks against more processor and to tailor the systems performance to a particular application.

Like the early models of the Cray T3E, the DEC 8400 systems use a DEC Alpha 21164 Microprocessor, clocked at 300 MHz. The processor is capable of issuing two integer or two floating point instructions per cycle, provided the instruction mix works out perfectly. The memory system design of a node in the DEC 8400 is based on a 3-level hierarchy of caches (L1, L2, L3) on the processor board and a shared dynamic memory (DRAM) on separate boards. The first two levels of caches are integrated on the DEC Alpha 21164 processor chip. The L1 cache is just 8 KByte in size; it is a simple, data-only, direct mapped, write through cache, but the access latency is only two clocks (i.e. 6.6ns). The L1 cache can supply two operands per cycle, yielding a peak read bandwidth of 4.8 GByte/s. The L2 on-chip cache is 96 KByte large, it is a 3-way associative unified instruction/data cache and has a write-back latency of 6 clocks (a peak bandwidth of 2.4 GByte/s). The write-back L3 cache comprises 4 MByte of fast SRAM (10ns) and is located on each processor board. According to specifications in [31] a 20 ns latency and a 915 MByte/s bandwidth could be realized.

The DRAM memory of a DEC 8400 is built from multiple boards with memory modules, which are two-way interleaved. With four memory modules, a maximal interleaving of 8 is possible. The vendor lists 176ns–928ns as an average latency for load operations from main memory, depending on how many memory-modules (i.e., memory banks) are installed and how many processors compete for memory access. The system used for the measurements in later chapters contains four memory modules with a total of 4GByte. Just like a single processor workstation, the DEC 8400 supports virtual memory. This type of a bus based multiprocessor system is depicted in Figure 2.6.

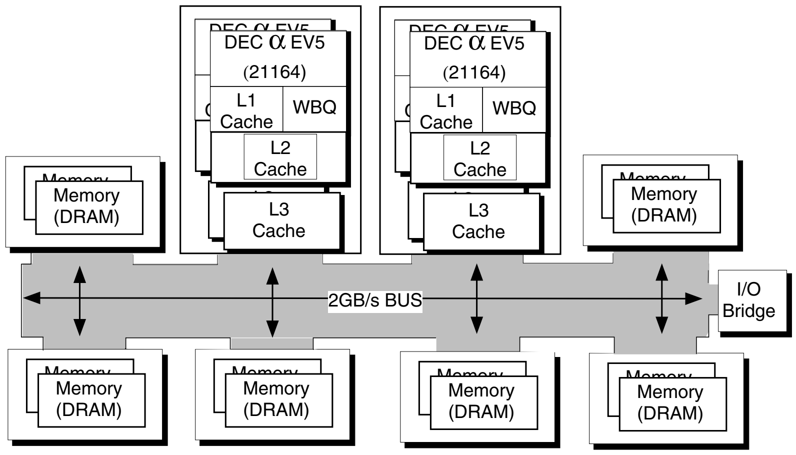


Figure 2.6: Overview of the DEC 8400 system architecture. The "communication" system comprises a high speed bus with several slots to incorporate processor, memory and I/O modules.

As a symmetric multiprocessor with fully coherent shared memory in hardware the DEC 8400 is more than a simple message passing machine. Its inclusion in my thesis serves the purpose to show that shared memory machines are prone to the same or to worse performance limitations than distributed memory system once they are used with message passing libraries.

References: [29, 30, 39, 31]

2.2 Network interconnects

This sections discusses the different high speed interconnects used for my studies of direct deposit message passing system. Among the example are meshes of fast links, a single high speed bus or even a large number of slower links organized in a clever scalable topology, such as a fat tree. All of them have in common, that the aggregate throughput can reach several Gigabit per second. Conventional networking interconnects, running on standard protocols like TCP/IP, IPX or X25 over Ethernet or ATM operate at much lower speeds and are used under different conditions. For those slower, but much more flexible networks the APIs of choice are TCP streams (sockets) or RPC stub generators. Message passing libraries that operate directly on the hardware are rarely used on this class of slower networks. A brief survey will explain the important characteristics and the properties of high speed interconnects suitable for direct deposit message passing.

2.2.1 Basic properties of a network interconnect

Unlike most traditional wide and medium (wide) area networks (e.g. IP or ATM switched networks) nearly all types supercomputer backplanes provide link level flow control mechanisms and error detection at the lowest hardware level to guarantee reliable transmission between the routing nodes. In most those mechanisms also work properly for multi-hop connections and under certain conditions these guarantees can be extended to connections between arbitrary endpoints.

For ATM such mechanisms are currently under development, but they seem to imply some complex network control architecture dealing with quality of service issues based on virtual connections and requiring the advance reservation of fractional link bandwidth and buffering resources. In a complete graph network the link level flow control can directly be used as end-to-end flow control, but complete graphs are not scalable for massively parallel machines and therefore highly impractical. For more realistic, scalable interconnection fabrics (e.g. meshes including all k-ary n-cubes or also fat trees) the link level flow control alone is not a suitable tool to so satisfy the control and synchronization requirements of most application programs.

In a realistic network, that uses link level flow control for end-to-end purposes some unrelated messages will eventually share a communication channel and those messages may potentially block each other through a link level flow control mechanism. Therfore such blocking makes it extremely difficult to write correct applications as long as they rely solely on the network mechanisms for their end-to-end communication scheduling. The best known solution to this problem is to prohibit the use of link level flow control for the synchronization of computation and require that all receivers *continuously and unconditionally remove* incoming messages [28]. Under this assumption most supercomputing networks can successfully rely on link level flow control hardware for guaranteed, error-free transmission while remaining entirely free of routing deadlock.

The interconnects of conventional global and shared memory systems can guarantee this type unconditional processing at the end-point easily, because the “receiving” operations are restricted to simple memory operations. Memory operations at remote nodes can always be executed immediately by a local memory module without the involvement of a processor or at least without interference with the control flow of the application program (if firmware handlers are used). More precisely there might be a simple dependence relationship between requests and responses, or between a directory lookup and the request itself, but those dependencies are usually broken with a network supporting multiple virtual networks to separate the different types of traffic (e.g. home location lookup (for a cache line), read requests and reply). For the smaller symmetric shared memory parallel machines a split transaction bus with a few levels of arbitration priority fulfills this communication task cheaper and faster than a traditional packet switched interconnect network. However buses will always remain a non-scalable interconnect structure and therefore be limited to smaller machines. Hierarchies of buses are subject to the same problems and limitations than the general interconnect structures.

Message passing over TCP/IP or TCP/ATM switched networks

Traditional wide and medium (wide) area networks (i.e. IP switched or basic ATM networks without service guarantees) do usually not include any hardware flow control on their links and overflow or congestion problems result in a loss of packet in the intermediate switches and require end-to-end retransmission of lost data. Protocols capable of detecting loss and of retransmission necessarily involve state at both ends of a data transmission. Because of this state they are said to be *connection oriented*. In general a connection oriented network architecture does not fit message passing libraries very well. Still many public implementations of the popular message passing APIs (e.g. PVM and MPI) can operate using reliable TCP streams as their means of communication, but their performance is limited by a fundamental mismatch of their model and the underlying communication structure. In such implementations a huge number of TCP connections must be kept open to route messages directly between all nodes of a massively parallel system. In several operating system the number of simultaneously active sockets is limited to 64 or lower. To work around such restrictions the connection based message passing libraries must establish and tear down the connections on demand, hereby limiting latency and bandwidth of the communication services offered.

The Cray T3D

The interconnect topology for data transfers in the T3D is a three dimensional torus with dimension-order wormhole routing. Service and IO nodes are inserted into the regular grid in at least two dimension, so that they can be reached by dimension order routing without any problems. Routing is fully deterministic and determined by a global hardware routing table loaded at boot time. There are several sets of virtual channels to permit full torus routing with a dateline and also to permit a complete separation between operating system traffic and user program traffic. Jobs in the T3D are space partitioned and there are hardware registers that limit the routing of user data traffic to a particular partition.

The raw throughput for user data over a link is 150 MB/s and is the same for all links in the 3-dimensional machine. This throughput figure can be reached with a stream of packets of only four 64bit words and does does not include routing headers or remote address information. Therefore I estimate the effective data rate on the wires to be nearly twice that speed. The average round trip latency is on the order of 1 microsecond even on large machines. This latency is almost independent of the distance in the network. Measurements suggest 15-20 ns per hop, with less than 16 hops maximal distance on all publicly known T3D installations.

However two T3D processor nodes share a single network access. The number of network nodes is only half the number of microprocessors (or processing nodes). If just one of the two processor is communicating at a time, the network can be accessed at up to 125 MB/s, if both processors are communicating the full speed of a network access is available and each processor obtains 75 MB/s access bandwidth. From experiments with nearly optimal all-to-all communication I conclude that the network nodes themselves are capable of switching packets at full speed on all six links of the 3D structure i.e. with 6*150 MB/s.

Control transfers such as notification barriers (Eureka) and normal barriers are performed on a entirely separate control network organized into partitionable and-or combining trees. There are several independent wires, that are time multiplexed further to permit 16 independent barrier sets for the user and 16 sets for the system. The total number of barrier sets is managed as a system global resource that has to be partitioned among multiple tasks executing in space shared mode. With clever partitioning of the tasks and the networks this number can be increased. Due to this special hardware any partition in the Cray T3D can fully synchronize with a barrier in about one microseconds, a time span comparable to the single round-trip of a data message.

The Cray T3E

The interconnect of the Cray T3E is highly similar to the interconnect of the Cray T3D. However the network access is no longer shared by two processors and the sustainable bandwidth per link appears to be slightly above 350 MB/s, leading to an estimate for the raw link bandwidth of 600 MB/s. Routing is said to be adaptive, but the disadvantages of adaptive routing seem to be well hidden from the programmer.

The system does no longer perform synchronization on a separate network, but performs all control transfers (including barriers) on the main 3D torus interconnect. The network interface still includes the combining hardware for barriers, so that they can be processed without the involvement of the microprocessor. The time for such a barrier remains about the same as in the T3D although the performance of the other network functions increased by a factor of 2-3.

Many design details of the T3E network are either not generally known or were obtained under non-disclosure agreements preventing to the author to disclose them I hope that in retrospective some publications of Cray Research will fully explain the excellent performance data measured and described in the subsequent chapters of this thesis.

The Intel iWarp

The physical topology of the iWarp interconnect is a 2D-torus. Each VLSI component can be connected to four other components by a bidirectional link. Due to clocking considerations (the X links work one one clock edge the Y links on the other edge) the topologies are restricted to 2D meshes and tori. In the book on the iWarp project those links are called buses, although they are unidirectional and do not have a multiple access properties that buses normally have. Each physical link can transfer data with 40 MB/s. The combined latency of single switch/link hop is between two to four 50ns clocks depending whether the link is in far or near neighbor mode (1 or 2 bits on the wire at a time) and whether the route turns corners at the switch or not.

Messages are routed through this interconnect structures on pathways of concatenated logical channels, as I would call the unified architectural support for long and short lived connections within the routing fabric formed by the iWarp components. Each of the four physical links can be flit multiplexed among up to 20 logical channels. Flit multiplexed channels are not called virtual channels like in ATM systems, but rather logical channels, since their number is strictly limited to 20 per routing node. Each logical channel occupies dedicated buffering- and flow control state stored on both sides i.e. within the two VLSI components at either end of a physical link. The dedicated registers comprise an eight word FIFO queue, several flit scheduling control registers, some destination selection registers for flit forwarding necessary to concatenate pathways and also a flow control counter for credit based flow control on each logical channel. Multiple logical channels that share the same physical link by partitioning the bandwidth of physical link in an almost fair manner among themselves [68].

This extensive communication hardware support can be used to implement several different communication models including permanent connections for systolic computing [53], switched sets of connections for computing on a reconfigurable network [36] and - particularly relevant to this thesis - for short lived connections managed by a message passing library.

Logical channels can be concatenated into pathways without the involvement of the processors at the intermediate nodes. In our system this is achieved under the control of tagged header words prepended to the payload of a message. The hardware is designed for deterministic routes, that are completely controlled by the source and the standard technique of peeling header words at each intermediate routing stop is used.

For my message passing systems I developed several different routing modules. For a first routing module, some pathways were set aside and configured for some auxiliary static connections. In most applications I run a dedicated static network for the operating system and a dedicated network for barrier synchronization. With some tricks this uses only 4 logical channels per node. The remaining 16 logical channels must then be assigned statically to the four directions of the physical links. With 2 and more logical channels per direction a router module can assign multiple logical channels to one direction and permit a second lane for messages to pass each other, but for this the in-order delivery requirement must be relaxed. The performance increase was studied in a student project [103] but the effort had to be abandoned after discovering the *left turn from passing lane* bug in the VLSI component. For the second routing module multiple logical channels were used in two allocation pools. Similar to corner turns a peel-off routing header is used to make messages switch the pool of resources at a given point in the network. With pools logical channels can be grouped into a two virtual switching fabric permitting the definition of a dateline and as a consequence deadlock free routing on a torus. I implemented a torus routing as part of my direct deposit message passing system but could not observe the improved throughput that are promised by the theoretic studies. I never found out why, since real iWarp routers are implemented in hardware and unlike the simulators that delivered the dozens of routing papers, the real hardware never provided complete insight into what happened within the router at full congestion. I suspect that some implementation problem in the VLSI component is responsible for this discrepancy between theory and practice. Therefore a simple surface routers is used in all measurements of this thesis. It delivered about one fourth of the maximal sustainable bisection bandwidth, that was proven in an static personalized all-to-all communication (e.g. a transpose of a 2D array) running under perfect congestion control [55, 99].

The routing fabric formed by the logical channels of iWarp is fully reliable, in the sense that errors are detectable through parity and that a concatenated channels incorporate flow control that works well from end-to-end. Using link flow control end-to-end requires careful attention to deadlock properties like blocking between unrelated routes. The iWarp interconnect is a therefore a typical supercomputer interconnect that works fast and reliably, but in turn needs more global assumptions than interconnects of the Internet world.

The Intel Paragon

The routing network of the early Intel Paragons is similar, but not as sophisticated as in the network in the iWarp system. The design of both interconnects have some common roots in the Caltech Cosmic Cube project [91]. The second generation Paragon systems adopted many of the communication features of the iWarp in their Cavallino router, since the communication system designers of the iWarp team joined the Paragon effort after completing their work on the iWarp VLSI component. However the experiments of this thesis are based on a first generation Paragon system using the same routing strategy and infrastructure as the Touchstone DELTA prototypes. The network of those first generation Intel systems is simpler than in iWarp, but faster. The peak speed of a physical link is up by a factor of five at 200 MByte/s. The network must be organized as a fully populated mesh and there are no back-loops to for torus routing. So a simple surface router with one single corner-turn is sufficient and the only choice for routing messages from any point A to any point B in the Paragon network. The routes are fixed and implied by the node addressing scheme. There is a broadcast facility that can deliver a message to every node in a rectangular sub area in the mesh but to my best knowledge it remained unused in the message passing libraries of both known operating systems for the Intel Paragon.

Contrary to the iWarp, the routing functions and the network interface of a Paragon are not integrated on a single VLSI component. There are separate NIC (Network interface) and MRC (Router) chips. In the iWarp architecture there is only one chip that does everything from processing to routing.

The DEC 8400

In this bus based, symmetric multiprocessor both, local memory accesses and inter-processor data transfers involve the bus-based high speed communication system. The DEC 8400 is built around a high speed system bus with 40-bit address and 256-bit data path. This bus is clocked at 75 MHz, a quarter of the clock frequency of the microprocessor, yielding a peak transfer-rate of 2.4 GByte/s across the system bus. This limit is reduced by arbitration overheads to a peak of 1.6 GByte/s under the best burst transfer protocol [39, 31]. A bus system puts limitations on scalability (fixed number of slots for processor and memory cards) but provides free broadcast, which significantly simplifies the implementation of globally coherent caches.

Split memory system transactions permit the optimal utilization of bus bandwidth when multiple processor communicate simultaneously with multiple banks of memory. Fairness in the arbitration protocol is required for deadlock free communication services across the bus, but such solutions are standard in all bus systems used in todays SMP super-workstations.

With a total capacity of 2.4GB the impact of the bus topology of the communication system is probably insignificant for the implementation of message passing libraries on the DEC 8400 SMP, but there are clear bottlenecks of memory system load when multiple processor contend for memory access.

Chapter 3

The anatomy of a message passing system

3.1 A brief history of message passing communication

At the early ages of parallel and distributed computing every vendor supplied a different custom message passing library with its massively parallel machines. As a result, the first generation of message passing systems were written. Among those libraries are the NX library by Intel, the CM5-MP libraries by Thinking Machines and the corresponding libraries by MassPar or by nCube Systems. A few independent software vendors tried to promote portable message passing programs and developed machine independent message passing libraries, available for several parallel machines and also for networks of workstations or clusters of PCs interconnected by common TCP/IP technology. The portable and vendor independent libraries are limited to the greatest common divisor of all functions supported in the variety of distributed memory machines and most of them suffered from sub-optimal performance.

A bigger standardization effort started within the National Labs, the sites of the big message passing users. A first standardization effort was lead by Argonne National Labs together with the German GMD and resulted in a few macros that mapped the messaging primitives of different platforms into a standardized message passing API. Of course this attempt was somewhat simplistic and naive, since macros could only even-out syntactic differences between the different native message passing systems, leaving all the semantic differences between the implementations as open pitfalls. The GMD/Argonne standardization effort was followed by the creation of the Parallel Virtual Machine (PVM) at Oak Ridge National Labs/University of Tennessee and finally lead to the broader standardization effort in the MPI forum, that was supported by a consortium of users and vendors.

The process of creating a good message passing library was seen as a standardization effort and the core problem, the semantic differences of message passing models received relatively very little attention. An rather theoretical definition of the operational semantics found in real libraries was attempted by [27], but its detailed formalization remained to messy to have an impact on the real word. Freely distributed public domain implementations played a crucial role in the acceptance of the standards in the industry. Besides a few free widely distributed implementations that ran on almost any communication system, some vendors chose to implement their most efficient production systems according to the new standard API or a subset thereof. In particular Cray Research shipped their T3D machines with PVM 3.0 as their best vendor supported message passing system. This message passing system rapidly became the most popular way to programming a T3D although CRAY tried to push its vectorized data parallel paradigm with CRAFT instead. CRAFT stands for Cray Fortran, a generic non-standard High Performance Fortran compiler that came to life far behind schedule. For most of the T3Ds life-cycle Cray PVM 3.0 remained the only other officially supported tool to write parallel programs for the T3D, with some users even escaping to some partially supported low-level primitives. In a similar way IBM choose to implement MPI as primary message passing system for its SP/1 machines. IBM had only few customers who migrated from vector codes to their MPP products and therefore IBM did not push HPF style data parallel programming as much as Cray did. With strong support from IBM the MPI libraries became increasingly popular and have superceded PVM in all newer application codes.

3.2 Overview of the software components in a message passing system

All messages passing systems mentioned in this paragraph contain a set of common, basic components or modules. I am using the term module and component interchangeably and without reference to their precise definitions in object oriented programming, because the domain of this thesis is not a software design methodology, but rather performance and software efficiency. According to my understanding a message passing system is simply a piece of low level communication software and its software components define and implement some communication protocols. In the networking literature such systems are classified and documented in layers according to the OSI reference model. Most message passing systems use some dedicated high speed communication hardware to support functions and protocols at various levels. The common OSI notion that lower level protocols are in hardware and higher level ones are in software does help much to understand gigabit per second networking, since the OSI model was created for common inter-networking and not for highly specialized and optimized implementations. Message passing systems remain specialized to applications of high performance computing and therfore a strict layering according to OSI remains hard to accomplish. So for the sake of simplicity, my bottom up discussion of the message passing software remains informal and omits the usual reference to the corresponding OSI services and protocols.

The lowest level components in a machine specific message passing library is a *driver* for the network interface. Another networking component implemented at almost the same low level is a *router* or a routing table to determine a fixed route between two endpoints in a given network. Typically such routes are calculated or looked up separately for each data transfer since in the most general case, message passing can be packet switched and there is no mandatory, permanent connection along a fixed route. Many parallel computers are built with a reliable interconnect and therfore the two software components router and driver are sufficient to implement a simple message passing system with a basic API featuring send/receive calls. Libraries with just those two components are limited to move data and there is no additional support for buffering or proper end-to-end flow control. In this context programs must adhere to some restrictive model semantics to use such a simple communication system. For the approximation of a more general messaging semantics (e.g. the postal model) the message passing system must provide separate end-to-end flow control or extensive buffering along the forwarding paths (i.e. store & forward or cut-through routers). A more precise notion of the message passing models and their semantics are given in Chapter 4.

Message passing libraries that fully support advanced models must include a *buffering module* in addition to the router and a driver. For a symmetric shared memory multiprocessor, the message passing library can import all functions for data transport directly from the hardware logic that is responsible to deliver coherent shared memory and therfore those implementations require just a buffering module in software. The similarities and differences of various buffering schemes and their impact on the API are discussed later in this section.

The three principle components (driver, router and buffering module) are sufficient to implement a bare bone message passing library with a simple API and some clean postal message semantics to its users - at least as long the message volume stays within buffering limits. Still, most message passing systems, like PVM and MPI, offer many more functions beyond the core API of simple messaging service with send and receive calls.

PVM was conceived to provide a virtual parallel machine to the programmer. Therefore the API includes calls to spawn parallel tasks and create different threads on an SMP or different processors on an MPP. The collection of functions for spawning, migration and awaiting multiple threads is called a *task management* module. The precise functionality of this module differs widely and portability of programs may still be affected by the underlying OS and the OS services required. Some message passing libraries require a fully functional network file system to take care of the distribution of binaries and none of the commercial systems I encountered so far, could deal with true capabilities for secure data access (e.g. manage AFS or Kerberos tokens). Several MPPs do not use the task management module of their native message passing libraries. Their runtime system provides an independent processor numbering scheme and a simple SPMD (single program, multiple data) model of execution with an implicit process spawning at the load time of a parallel binary. Cray PVM 3.0 is such an example.

Another common extension of basic message passing calls are global communication primitives. Those functions can be labeled as the *collective communication* module. The simplest global communication primitive is a barrier synchronization, which is a simple reduction operation without data - followed by global broadcast. The operation serves as a partial or global synchronization primitive for the program flow. Further collective operations are reductions and broadcasts with data items, scans, gathers, scatters, and personalized all-to-all communication. A simple implementation of those operations can be done by composing basic send- and receive calls and their implementation could be left to a supplemental library. Modern message passing systems postulate the collective communication operations as primitives, since there are many known tricks to exploit the regularity of the communication patterns for better communication performance (e.g. congestion control or message combining) at the lower levels of the communication system. MPI has a particularly rich set of collective communication primitives.

There is even more functionality that could be incorporated into a full service message passing library. One area still under investigations is *parallel I/O*. Research on parallel I/O includes some aspects of processor naming, efficient message routing and collective communication, during the redistribution of the between compute nodes and the I/O nodes of a parallel system.

In my thesis I would like to concentrate on the first three components of a message passing system and provide more insight about the right functionality for best possible software efficiency and best performance. Therfore I am considering the aspects of task management, collective communication and parallel I/O to lay outside the scope of this thesis.

3.3 A survey of the core components of message passing systems

In this section I present a brief description of the three core components of every message passing system, the router, the driver and the buffering component.

3.3.1 Messages routers

Contrary to a common belief shared by many members of the theory community, the sophisticated algorithms for congestion control or for adaptive routing do not determine the overall system performance in most practical message passing systems. The algorithm itself is less critical but it remains most important how the properties of a routing method are incorporated into the overall software system. A brief discussion states my experience with different routers and explains how adaptivity and and too much cleverness in the router can make message processing at the end-points extremely difficult.

Fixed routes in hardware

The problem of routing in early parallel machines was solved by the hardware designers. The routers of the Delta Touchstone and the Intel Paragon maneuver the data packets from source to destination in highly deterministic manner along a hardwired path. Routes are limited to a single corner-turn and are determined by the hardware limit registers. This routing strategy works on the surface of a mesh, but it is even too simple to permit deadlock free torus routing or non rectangular machine configurations. Despite all limitations the router was successful since it enabled a fast network and guaranteed the delivery of packets in order between any pair of nodes.

Adaptive routes in hardware

An good example for an adaptive router is the Thinking Machine CM5. In its normal mode of operation of router determines all its routes in hardware. The fat tree topology requires a two phased routing process. In a first phase, the packets are routed upwards to a common ancestor node. In this phase the hardware can use adaptivity and randomization to balance the load and avoid congestion on certain links. Once the messages arrive at a common ancestor node they follow a deterministic route downwards to reach the proper destination node. In the normal case, such routing is done in hardware and the network is completely reliable due to built-in flow control and error detection. However the adaptivity and randomization in the router destroys the usual in-order delivery property, that is granted in simpler networks. The side effect of the clever CM5 routing algorithm makes the implementation of coherency protocols quite difficult and in the CM5 message passing library every single packet has to be accounted for and acknowledged separately.

Source controlled routes

Source controlled, fully deterministic routing schemes are used in the Cray T3D and the Intel iWarp. The structure of those routers is quite simple and with appropriate link level flow control the interconnects can easily provide a fast, reliable in-order delivery of messages on along a given path from node A to node B. As their principle drawback, such simple routers rely on some global assumptions about deadlock. Their static rules of layout becomes a problem in case of failures and prevents adaptivity to avoid congestion. Defective nodes can still be mapped out, but only at the cost of a global update of the routing table in the entire system. Such behavior does not sound very appealing in theory, but in practice it is quite compatible with the mode of operation of supercomputers with restartable applications jobs. More sophisticated routing systems have been proposed, designed and even built, but it is extremely hard to find system software that can take advantage of their capabilities. So I am not aware of any viable, practical alternatives with the simplicity, robustness and cost effectiveness comparable to source controlled routers.

3.3.2 Connection based routers

For connection based networks (e.g. ATM) the routing of frequent single messages between a large number of processors becomes a two tiered routing problem. At the lower layer, the network itself will establish connections and route data packets internally along the paths of those connections. Routes are determined at connection setup time. Typically setting up connections is quite expensive and can not be done for every little message or ATM cell. The nature of a connection implies that there exists some state and resource reservations associated with each of those connections along a fixed route. Examples could be are flow control state, cached routing information, reserved buffer space and virtual channel entries in protocol field. The reservation of resources along the path typically limit the number of connections that can be held open simultaneously. For example most TCP/IP socket implementations are subject certain limitation on UNIX systems that permit only a maximum of 64 open files. Given the limitations at the lower layer, the message passing library must either store and forward the messages between certain routing centers or start to manage connections appropriately. Managing connections is clearly a function provided by the routing module of a message passing system.

Since most MPPs do rely on networks with reliable datagram services, the strategies for implementing message passing over connections is beyond the interest of this thesis. The considerations of this paragraphs are inserted for completeness and none of the experimental platforms used in the the rest of the thesis do rely on connections for their basic message passing services.

3.3.3 Network drivers

The most challenging aspect of a network interface driver is an inherent two threaded architecture. Unlike a display driver that moves data to a device at high speeds or a video capture driver that received data from a device, a good network interface driver is entirely symmetric and must be capable to send and receive information simultaneously at full speed. While the amount of state in a disk subsystem or a graphical co-processor is relatively small and the standard transactions are quite simple, the amount of state in a typical network is almost unbounded and the considerations of permissible state transitions in networking (e.g. for deadlock avoidance) are highly complex.

Because most parallel programs have a perfect balance between sends and receives, the two activities of a network driver are equally challenging. Good performance and good software efficiency is only achieved when both activities make progress simultaneously at their maximum speeds possible. The nature of message passing programs implies unordered sends and receives and the network drivers must be able to handle them. Therefore message passing drivers are at least two threaded and in most systems they share a common memory system and a common processor. In applications with overlapped communication and communication the message passing system further shares those resources with the computational units working on the compute part the application. The scheduling of resources used by a message passing system can be done in several different ways:

Polling network drivers

Every network interface contains some status flags to notify the connected processor of any state changes in the network. Examples of such a state change are “arrival of a message” or “network ready for another message”. Typically the network status flags assert that a particular action can be performed safely. Those actions can relate to a full or a partial message. In the first case, there are flags to indicate that it is safe to “send an entire message” or “receive an entire message”. Systems with support for short messages (e.g. the CM5 Network) typically permit atomic actions on an entire message. In the second case, the actions are more primitive and some example actions are “set a route”, “open a message”, “transfer a few word to a FIFO”, “start a DMA engine”, “close a message”. The Intel iWarp and Intel Paragon Systems are good examples of the second kind of message handling and both of them have a highly complex model of operation to send and receive long messages safely. The real implementations of the the two latter communication systems are prone to many design mistakes and implementation glitches that can push even the best system programmer to his/her limits of understanding and handling the complexity of the state machine involved (several flags of the Paragon B-step NICs were caught lying under certain well defined but unrelated conditions - coding a correct driver resembled more to a gaming session of an adversary strategy game than to a programming effort).

Without multi-threaded CPUs the structure of a polling network driver resembles HCI code with a big event loop to poll all the different conditions and state transitions of the network while assuring progress at either the sender part or the receiver part. On most systems, polling flags is faster than dispatching interrupts even if they are dispatched directly to a low level driver. However some networks have a requirement that arriving messages must be served within an small, fixed amount of time that is much smaller than a time slice of a typical operating system. Once the same node processor is also used by the application code for computation response time guarantees are hard to implement with polling alone. Therefore many network interface drivers support interrupt driven operation as well as polling, using polling when network activity is high and interrupts when network activity becomes low and the attention of the system shifts to computation. The most common event linked to an interrupt is "message arrival".

Interrupt driven network drivers

May networks require that incoming message are served immediately. Linking an interrupt to message arrival is therefore a common option. While conceptionally simple and convincing dispatching interrupts at every message arrival must consider the granularity of communication. Dispatching an interrupt involves task switching to a handler and costs time, especially on a heavily pipelined modern microprocessor. Therefore many systems try to use interrupt coalescing by collecting several incoming messages before an interrupt is generated and then handling multiple incoming messages at the same time. The typical Ethernet network interface collects several messages before generating an interrupt. This strategy reduces the overhead for interrupt processing but increases the latency of an isolated message, since the detection of an arrived message is delayed until several messages arrived or a time out is reached.

The potentially large overhead for interrupt dispatch into the user code prevented the effective use of buffered user messages between node programs on a Cray T3D, although the hardware support for them was provided. For Intel iWarp and Paragon systems messages can be quite large and interrupts are even used to receive or transmit parts of a message. In particular once the message body is handled by a Paragon DMA or an iWarp spool the completion of the transfer is often flagged with an interrupt.

For fast reaction in efficient message passing system interrupts must be handled at the lowest possible level (i.e. in the kernel) and be highly optimized. The dispatch time on iWarp was improved by optimizing the register usage of the handler and even by limiting the size of the handlers for optimal I-cache usage when the processing alternates between send and receive functions.

3.3.4 The buffer management

The buffer management strategy of a message passing system can be classified according to two most important parameters. The first parameter is the location where the buffering takes place. The second parameter is the provider of the buffering space.

Buffers for entire messages can be placed in several locations along the forwarding paths in the end-points and the network. For store-and-forward and cut-through routing networks buffering can further take place in some intermediate nodes along the path, but for our most common case - the wormhole routers - the network itself does not have the capacity to buffers entire messages, so in wormhole networks the buffering options are limited to *sender buffered* or *receiver buffered*.

The management of buffers is a simpler choice. Message buffers can be allocated automatically in system space or they can be allocated explicitly by the application program in user space. In Section 4 I will discuss the relationship between buffering strategies and the semantics of the message passing API. For an introduction I briefly summarize the different approaches taken by a few known message passing libraries.

Sender buffered systems

Sender buffering is most common choice in message passing implementations for symmetric, fully coherent shared memory systems such as the DEC 8400 or the SGI Power Challenge. In those systems the mechanism for moving data between processes and processors is hidden behind the coherency hardware and works on a pull basis in most instances. With a fully automatic shared memory there are no mechanisms to indicate the destination for a data transfer except for the global address in main memory. Those systems are built to exploit temporal and spacial locality automatically and therefore the data stays closely to the sending processor until another processor acts to receive and access it. So the natural choice for the allocation of the buffers remains on the sender side and it happens best with the invocation of the send call. The receive side is responsible to notify sender side when the data is received, copied and passed along to the application on the receiving side.

In systems without global address space or in systems with a better control over push or pull, sender side buffering is often chosen for large messages. For large messages the Intel Paragon NX message passing library [85] chose to buffer the data on the sender side and pull it only upon receive, although there was no hardware support for pull. The implementation used a separate control messages and an interrupt handler to start the send of a large message. Small messages remained receiver buffer buffered because of the smaller overheads and better latencies. The sender buffered strategy has several advantages regarding processing overhead for buffer management and regarding handling of faults. The buffers are allocated as an action of the send call and the necessary processing power can be drawn directly from the application thread invoking the send call. This is in contrast to the case for buffering on the receiver side where the network condition of a pending message arrival triggers a receive thread that must deal with buffer allocation. The second advantage of sender buffering with implicit buffering is that the application can be notified precisely and accurately when the system is about to run out of buffering space. If buffers run out at the receiver the message must be discarded causing a global abort. [1]

3.3.5 Receiver buffered systems

For the message passing libraries of distributed memory parallel systems and network computers the receiver buffered approach is most common. Their network management strategy requires that arriving messages are removed immediately and unconditionally from the network. This requirement implies the presence of a receiver buffering module that provides buffer space for arriving messages, whose matching receive calls are not yet pending.

All implementations of PVM do buffer incoming short messages in a system buffer pool at every node. In PVM the size of that buffer pool is controlled by an environment variable. Upon buffer overflow some data is lost and error messages are printed to the console. Some libraries have sender buffering as well and the interaction between the two strategies is not transparent to the programmer. In some cases it helps to make messages larger so that the overall timing or buffering strategy changes for the better and a given application program that was out of buffers before suddenly becomes runnable. With the precise interactions of parameters and the limits on buffer space unspecified, the setting of those environment variables often becomes a black magic of tuning.

In this sense of buffer allocation my direct deposit message passing system will receiver buffered like most standard libraries. However the allocation of buffer space is not implicit, but application specific and always pre-arranged before the data transfer takes place. This simplifies buffer allocation upon message

[1]The basic guidelines for high performance messaging are a minimum of copies and simple protocols. Therefore the introduction of a retransmission protocol implementation like TCP/IP is unacceptable in many cases should only be used as a last resort.

arrival and permits the specification of non-contiguous and complex target data structures to be used as buffers for the data.

3.3.6 System buffer allocation

The basic message passing systems and most early implementations of vendor specific message passing systems require buffer management as side effects of the internal send and receive operations. Therefore the data is taken over (semantically copied) from the user upon a send call invocation and data is delivered (semantically copied) to the user upon the return of the matching receive call. If a message passing library requires some additional buffer space to adapt to network availability, the library has to provide its own buffer space and copy or map the user data to its own buffers. A buffer allocation strategy that is carried out by the message passing system or the message passing library is called system buffer allocation. Note that the user - system boundary of the message passing system and the operating system might not necessarily coincide.

3.3.7 User buffer allocation

Application or user buffer allocation is enabled by split send and receive calls. Split calls partition a send action in a send-initiate and send-complete. The bracket of a matching send-initiate and send-complete mark the corresponding buffers as “in use” and prohibit further changes by the application. Similarly the bracket of a receive-post call and a receive-complete call marks a receive buffer as “in use” and ready for the messaging system to store received data.

In message passing systems with user buffer allocation the data buffers are allocated by the application using split-send and split-receive calls. Buffer allocation by the user is an option for buffer management by the sender or by the receiver. Sender-side buffers are allocated by the application just before the send-initiate and they are freed or recycled after the messaging system flags a send-complete. Receiver side buffers must be allocated by application as empty buffers before issuing a receive-post call and returned to the application as filled buffers upon a receive-complete call.

Summary

A small chapter on iWarp implementation 7 contains a few details about the implementations of a sender buffered and receiver buffered messaging system on that target machine. The evaluation can be summarized in advance: The disadvantage of sender side buffering is an additional complexity in the network driver. Sender buffered systems require the negotiation of an open path between a sender and a receiver before the date is being pushed into the network. A brute force push can be made deadlock free and reliable with some tricks, but blindly pushing data leads to bad congestion behavior in most wormhole routers. The drivers of a receiver buffered system are simpler to implement. Receiver buffered system can maintain correct, postal semantics (see Chapter 4) as long as every incoming message is promptly removed from the network. Buffers in receiver buffered systems should be allocated quickly and since the network is blocked during buffer allocation. Therefore the allocator at the receiver side must remain simple and efficient. An very interesting variation in the design space of receiver buffer messaging is taken in the Urbana Fast Messages for the Cray T3D. Buffer allocation is performed by the sender but with a remote memory operation in the address space and the memory of the receiver. This is achieved by some highly efficient atomic swap-and-update instruction that the designers of the Cray T3D implemented for synchronization [62].

The question whether buffer are allocated implicitly by the system or explicitly by the application is fully orthogonal to the question whether in a protected OS environment such buffers can be transfered between user space and system space without copies. The point of distinction is whether an action of the message passing system (such as the arrival of a message from the network) causes a buffer allocation or whether it remains entirely under application control.

Chapter 4

User models for the services of message passing libraries

A message passing subsystem is part of in a parallel or distributed system and transfers *data* from one process or thread to another. In the most general setting, such a transfer can be achieved by a wide spectrum of hardware mechanisms. Those mechanisms include interprocess communication between two time-shared tasks or threads executing on the same processor using the same memory system, transfers across coherent shared memory segments between truly parallel threads in an SMP or a message based communication operation between two separate processors and separate memory systems across an interconnect or a network.

Without loss of generality I am restricting this discussion to the vocabulary of distributed memory systems by assuming full parallelism with exactly one memory system, one processor and one process per logical node. For a message passing programmer the distributed memory the view is exactly the same for symmetric shared memory parallel systems or for pseudo parallel system with multi-threaded programs - just that in multiprogrammed system the operating system (i.e. the threads package, scheduler and the synchronization primitives) provides the illusion of a large number of virtual processors with lesser performance that can be programmed like if they were physical processors.

For message passing communication in this broader sense, there exist a variety of options to organize the transfers of control and data. In the simplest case of a data transfer, the transfer is initiated by the sender node $\mathcal{S}$, and the original location of the data is in the local memory of sender node $\mathcal{M}_{\mathcal{S}}$. The sender $\mathcal{S}$ invokes a *send* operation to transfer a *message* to the destination node $\mathcal{R}$. The destination node $\mathcal{R}$ invokes a *receive* operation to retrieve the message and to store it into local memory at the receiver $\mathcal{M}_{\mathcal{R}}$. At this point I do not want to consider non-determinism through split *send* and *receive* primitives or through wildcard parameters of *receive* operations. In the simple message passing model each message transfer the *send* must conceptionally match a single corresponding *receive*.

The originator of the transfer is not restricted to the sender node. With appropriate hardware or software support a destination node may initiate the transfer and *fetch* some data out of the senders memory, or vice versa the sender might *deposit* some data into the receiver's memory and even complete the whole data transfer without the receiver's participation. The difference between a classic *send/receive* and a *fetch/deposit* transfer is in the synchronization and the control transfer provided along with the data transfer. A description of message passing systems purely based on the mechanics of data transfers misses the important point of the synchronization that goes along with every data transfer. A data transfer can take place under a variety of synchronization assumptions and the programmer must be well aware of the control transfer semantics to write correct parallel programs that are free of data races and free of deadlock.

Every message passing library comes with a set of assumptions and rules to write correct programs. Such a set of a assumptions is called a *message passing model*. The two well established models of message passing are the *rendez-vous model* and the *postal model*. Those basic models can be defined with a single type of messages. For the extension of the basic models to the *deposit model* I require a precise distinction between control and data messages, as described in Section 4.1 of this chapter. The next sections describe the two traditional models of message passing and discuss their advantages and shortcomings. In Section 4.2.3 I am going to define the *direct deposit* message passing model, as new model that combines the advantages found in both traditional models.

4.1 Control and data transfer messages

For the model discussion in this thesis all messages are classified based on their content, their length and their purpose and are separated into three classes: control messages, data messages and hybrid messages:

- *Control messages* are inherently linked to synchronization. The transmission or reception of such a control messages does not move any data, but propagates a logical assertion between the sender and the receiver. The meaning of such an assertions is, that some data block is ready to be transferred, that some buffer is available to receive more data or that some previous data message was or was not transferred successfully to its destination. In the latter case the reception of a positive or negative acknowledgment automatically initiates retransmission or flags an error in a communication system that supports reliable transfers.

  All implicit messages generated by the lower protocol layers of a message passing library are classified as control messages. Even synchronization primitives such as global barriers are best viewed as a collection of combined control messages, regardless of whether a barrier is transmit over a regular data communication channels or whether it is performed by special purpose hardware. The sole purpose of control messages is to communicate program state and implied assertions between processors in a parallel or a distributed system.

- *Data messages* can contain large amounts of data. The amount of data moved by each *logical* message typically exceeds the hardware buffers along the communication path. The proper handling of data messages involves some local memory accesses at the sender and the receiver side to retrieve or store the data from or to memory the user program. Data messages require a door-to-door service, and the data must travel from user space of one process at the originator node to the user space of another process at the destination node. Regardless of the precise location of the data at the source and the destination, data messages always contain a large amount of data that is likely to exceed the capacity of buffers along the transmission path.

- *Hybrid messages:* In some cases the messages contain pure data as well as pure control information. Those are called *hybrid messages*, but for the subsequent models, I propose to classify and handle each hybrid message either as a data message or as control message, depending on the amount of data it contains and depending on the circumstances of its transmission. In the case that a hybrid message can not be handled cleanly in one or the other way by an underlying message passing system, there is always the possibility to decompose it into two messages, a data and a control message in the lower protocol layers.

  The motivation for such a strict classification of messages is the intent to use the message properties in some new, more advanced models of message passing, like the *fetch* or *deposit* models. Looking at efficient implementation of a messaging model, it can be shown that handling of messages can be simplified and accelerated based on their properties. Since buffering is the key problem of efficient message passing a classification of hybrid messages based on their size of their data field seems to be most appropriate.

According to my experience with a message passing programs for parallel systems, hybrid messages can be generated in two cases: (1) Many flow control and other protocol implementations bind some control information, such as an acknowledgments or flow control credit packets to data messages flowing into the other direction. In this case the messages contain substantial amounts of data and must be handled like data messages. (2) Hybrid messages also occur in certain global data parallel operations like scan-reductions. The global operations typically synchronize a computation with a global data dependency, involve only a couple of machine words of data and a limited amount of local computation when the message is handled. Therefore they can be handled like control messages. In many cases they are combinable and can be handled immediately by the communication system or a low level driver - without buffering or invoking user code at the intermediate nodes.

4.2 Message passing models

The specification of a communication service (e.g. a message passing library) defines some operational semantics of the various communication primitives involved. The message passing programmer or the code generator of a parallelizing compiler incorporates the semantics to insure proper operation. For example, a message passing service may guarantee in-order delivery of messages between any pair of nodes and the program might rely on this semantic property to insure correct execution. Two important properties of a communication system have an immediate impact on the correctness of parallel programs: the ordering restrictions of messages and the ordering restrictions of resources used for the larger data transfers. The first set of constraints is considered as a *coherency model* and the second set of constraints must be considered to guarantee *deadlock-free execution*.

The mechanics of data transfers used in a message passing library imply certain properties and deliver certain guarantees to the application program. I distinguish between different message passing models depending on which properties and guarantees are agreed upon between the message passing library and the program. Those agreements will be defined as properties of a *message passing model*. Most models of message passing derive some high level programming rules from the mechanics of the data transfer, like e.g. if a programmer uses a particular kind of send primitive, there will be certain limits of message sizes. Those rules aim to guarantee proper execution of a program, but for most libraries on the market, their low level of abstraction or lack of strictness and conciseness remain highly dissatisfactory. In the three following subsections, I will attempt some better definitions for the two most common models in their usual terms and also for my own deposit model by linking the mechanics of direct deposit transfers to a program property named *well-synchronized* programs.

4.2.1 The rendezvous model of message passing

The oldest and most rigorous of the existing message passing model is the rendezvous model (“le rendezvous”, French for appointment or meeting). In this model every data transfer must use a two-way handshake to ensure a proper meeting between the two communicating processors or threads for any data exchanged. The model requires total control over the communication schedule in the distributed computation. The simplicity and the rigor of the model made it popular for theoretical work, the foundation of the CSP [57] calculus and the implementation of the Occam [86] programming language. In its strict form the classic rendezvous model deals with atomic *sends* and *receives*. The conceptional work of the CSP as well as the practical implementations of OCCAM include non-deterministic send- and receive-primitives that allow an extension to other models than pure rendezvous, but without determinism they define or implement a pure rendezvous model. Split send- or receive-calls to a message passing library could be modeled as a form of non-determinism, but including the precise semantics of a realistic, practical buffer management into a such a model results in such a messy description that becomes hardly usable by a programmer - an attempt to formalize a real world message passing library with a formalism has been done in [27]. In particular the formalism is of little help describing buffer limitations and the resulting description remains purely operational and at a low level.

For a better overview I choose a high level description of my model by linking the transfer properties of a message passing implementation to the program properties required for correct execution.

Transfer Property 4.1 (Rendezvous messaging) *Each data transfer necessarily includes a two way synchronization between the sender and the receiver.*

Figure 4.1 illustrates the expected communication behavior of a data transfer under the rendezvous model of message passing. The ordering constraints between the intervals of the send region (between start_send and end_send) and the receive region (between start_receive and end_receive) are the most important part of the model. While both a send- or a receive-call can be the first action to initiate a data transfer, there remains a mandatory overlap of the both execution time-lines, i.e. a meeting, a so called rendezvous must take place. In most practical cases there is an almost simultaneous completion of the operation at the sender- and the receiver-side. The semantics and the ordering constraints of the operations involved are enforced by control messages (black arrows), that are exchanged in addition to the data transfer messages (grey arrows). Depending on the communication system available, a data transfer of a large data block can be a single large message or a number of smaller physical messages forming a single logical transfer. What counts for the model is that the entire transfer belongs to a single pair of corresponding send- and receive-calls.

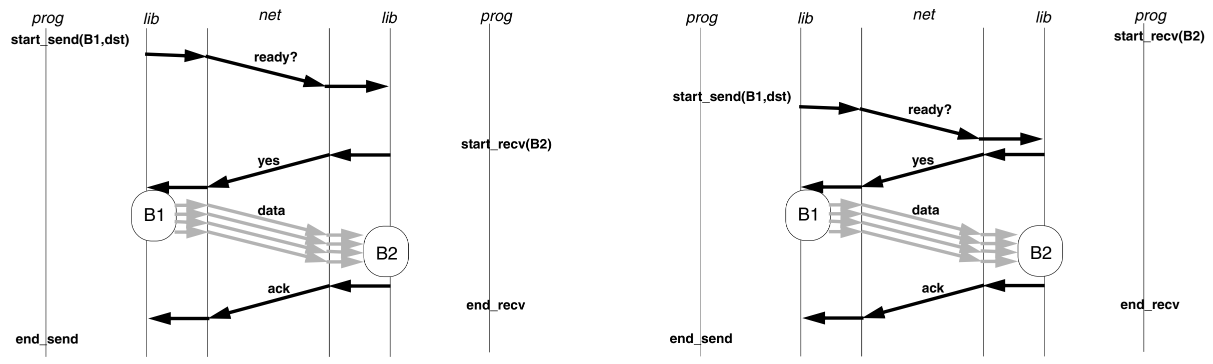


Figure 4.1: Expected communication behavior in the rendezvous model. Case when the sending processor enters first (left) and when the receiving processor enters first (right). The figure graphs interactions between two processors along a wall-clock time-line (top) to (bottom).

Program Property 4.1 (Fully synchronized program) *Message passing programs are said to be fully synchronized if all message transfers can be strictly ordered according to a global wall-clock time.*

The strictly ordered global time schedule guarantees that all matching send- and receive-calls are made in the proper order and all required meetings can take place as planned for by the global schedule. The definition of the transfer mechanics with a two way synchronization guarantees the atomicity of a data transfer. With these two rules we can infer the correct execution of a global time schedule and with it the correct execution of the distributed program.

In practice it is unrealistic to require a global wall-clock time for the correct execution of a parallel or a distributed program. With the classical arguments outlined by L. Lamport [70] the restriction can be weakened to partially ordered events on local time scales. A less restrictive and weaker program condition can ensure correct execution under a rendezvous model.

Program Property 4.2 (Strongly synchronized program) *A message program is said to be strongly synchronized if all related message passing transfers are partially ordered events (e.g. according to Lamport clocks).*

This relaxed, somewhat more practical program property still guarantees the proper invocation of matching send- and receive-calls and therefore can make a parallel program execute properly in the rendezvous model. This less restrictive program properties does already interfere with the possible implementations. The problem is the term *related* transfers. With such a definition the message passing passing model starts to make certain assumptions about the interconnect network. A general definition with partially ordered time between send and receives is only acceptable, if the assumed network forwards all messages independently i.e. free of routing deadlock. So any two message transfers must be able to execute concurrently and in any arbitrary order without interference through occupied common resource in the interconnect network. Most modern networks can give such a guarantee since they eventually deliver messages as long as they are immediately and unconditionally consumed by the receiver.

One of the drawbacks of the rendezvous model is the overly tight synchronization imposed on the send- and receive-process for the transfer of each data element. The restriction turns into an implementation advantage if only large blocks of data are exchanged as a basic unit. With a rendezvous in place the send- and the receive-process can both actively contribute to the transfer and exchange the data directly end-to-end without costly buffering.

Strongly synchronized programs for fixed communication patterns like, shifts, scatter/gather or all-to-all permutations are quite easy to device, but for arbitrary unknown communication patterns they remain hard to write. As an extension to the rendezvous model one could introduce buffer processes and model a message as several data words pipelined over a short-lived connection between two communication partners. In these extended model, a proper rendezvous meeting between sender and receiver is still established through a request-connection/accept-connection protocol, it is just that the unit of transfer is no longer a single data element. For large data streams the rendezvous style of communication raises questions of proper flow control along the links and challenges the capabilities of the network to provide long lived connections. With a limited amount of connections the model it can only accommodate transfers for restricted or carefully scheduled patterns (e.g. next neighbor or SIMD style communication). An example of carefully scheduled communication is balanced AAPC. Due to the regularity of this complete exchange pattern it is not difficult to find global communication schedules an therefore write correct programs compliant with the rendezvous model. Those programs can be highly efficient an reach nearly peak performance, since they are based on the raw, most efficient network primitives [56].

To illustrates the difficulties of programming the rendezvous model further, Figures 4.2 and 4.3 depict two examples of message passing programs. The two programs both involve three processors, and two processor (p2 and p1) are transferring one message each to the third processor (p0). The only difference is an additional message between the two sending processors (p2 and p1). While in the first case, in Figure 4.2, the program is *strongly synchronized* and therefore complies with the rendezvous model, this is no longer true in the second case in Figure 4.3. After the removal of "unrelated" third message it depends on the timing between the first and the second processor (p2 and p1) which message reaches the third processor (p0) first and depending on whether it matches the outstanding receive the program completes or deadlocks. With this race condition the program is no longer strongly synchronized and therefore does not longer guaranteed to execute correctly under the rendezvous model.

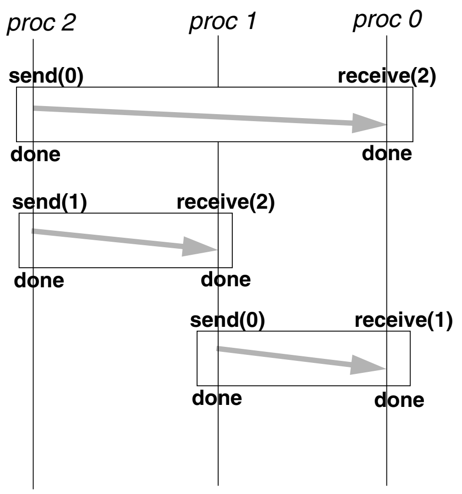


Figure 4.2: Example of a strongly synchronized program that executes correctly under the assumptions of the rendezvous model of message passing. The parameters of send/receive indicate the node i.e. send_to_node(1) and receive_from_node(2).

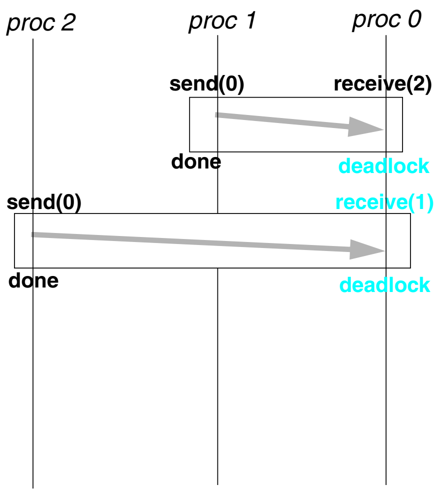


Figure 4.3: Example of a program that is no longer strongly synchronized and prone to race conditions under assumptions of the rendezvous model of message passing.

The reader familiar with OCCAM or CSP might notice that both programs given in Figures 4.2 and 4.3 will execute correctly on a multi-ported machine, because the example cases can all be properly resolved by the select statement in OCCAM. However, message selection capabilities assume hardware support for a direct physical link (or at least an exclusive virtual channel) from every node to every other possible communication partner. Such a direct and contention free communication channels are taken for granted in the PRAM [40, 88] model of multiprocessing. Unfortunately this model assumption is highly unrealistic for modern high performance networks with their powerful, but expensive communication resources that must be shared among different messages. Therefore the communication system must deal with contention for access ports, for intermediate buffers, and even for physical links on a per message basis. Such resource conflicts can usually be resolved automatically within the network but only in a non-deterministic manner. To the best of my knowledge none of the different flavors of the rendezvous models can appropriately deal with high performance data transfer and the non-deterministic delivery of messages to a single network interface port or other forms of blocking contention within the network itself.

In summary - the rendezvous model does not include the concept of buffering. If parallel or distributed programs need buffered communication in the rendezvous model, they have to explicitly implement and also model a FIFO buffer (e.g. with a separate buffer manager thread). The external modeling of buffered message passing is impractical and in lead quickly to the development of an alternative model of message passing, that includes the concept of buffering in the communication system or in the message passing routines at the end-points. This model is most often called the *postal model* of message passing.

4.2.2 The Postal Model of Message Passing

The postal model [7] of message passing incorporates buffering into the basic message passing services. Its definition and use is simple and highly appealing to the programmers and code generators. The receiver can receive the data immediately or at a later time. If the receiver does not claim the data immediately, it must be stored in the network or a temporary location (a mailbox) until the receiver is ready to accept the data. The presence of a mailbox and the features of the well known postal system to delay and store an almost unlimited amount mail in transit results in the intuitive name.

Transfer Property 4.2 (Postal messaging) *The postal model provides a one way synchronization with each data transfer.*

The consequences of a one way synchronization are obvious: No receive operation can complete before the corresponding data (i.e. the matching message) has been sent. But conversely the send can terminate regardless whether a receive is ever invoked or not. Buffering must take care of the situation of a send with a receive at a later time.

The postal model leads to a nice, much weaker property for correct programs that the previous rendezvous model.

Program Property 4.3 (Flow Synchronized) *A message passing program is flow synchronized, if it is written under the assumption that a sender S can send the data to a R receiver at any time, regardless of the state of the receiver.*

The program property "flow synchronized" is very simple and intuitive to the programmer, since the flow of data in parallel programs naturally synchronizes the sending and receiving computations in one direction. Unfortunately this elegance comes at a cost, the implementation of the pure form of postal messaging requires nothing but infinite amounts of buffer space, and therefore a correct implementation remains unrealistic. The postal model cannot limit the buffer space because a large message exceeding the available buffer space would force a sender and receiver pair into a two-way synchronization and cause problems. An accidental two way synchronization could result in either a program deadlock or a forced data loss after the receiver node discards the remainder of a message. The first problem occurs in the presence of a link level flow control protocol that throttles the senders when the receivers run out of buffers so no data will be lost, but a potential deadlock is introduced if a set of messages exceeds the buffer space available to them.

Figure 4.4 illustrates the expected communication behavior of a transfer under the postal model of message passing. There are no ordering constraints between the the intervals of the send region (between start_send and end_send) and the receive region (between start_receive and end_receive), except that a message must be sent before its receive can complete. Otherwise the programmer can send a message early and receive it much later or alternatively invoke an early receive and wait until the matching message arrives. There is no mandatory overlap and a send can complete well before a receive is even started. No control messages are required to enforce the semantics of the postal model.

Most standard message passing systems, such as PVM, MPI, and NX, pretend to offer a postal model at first sight, although MPI clearly states that implementations can offer buffering to some extent but do not have to. The other implementations reveal limits on buffer space a closer look and leave it to the user to deal with buffer overflows. In this postal model the semantics of the common cases (provided there are sufficient buffers) are easy to handle for a human programmer and postal messaging code is quite portable between different parallel systems.

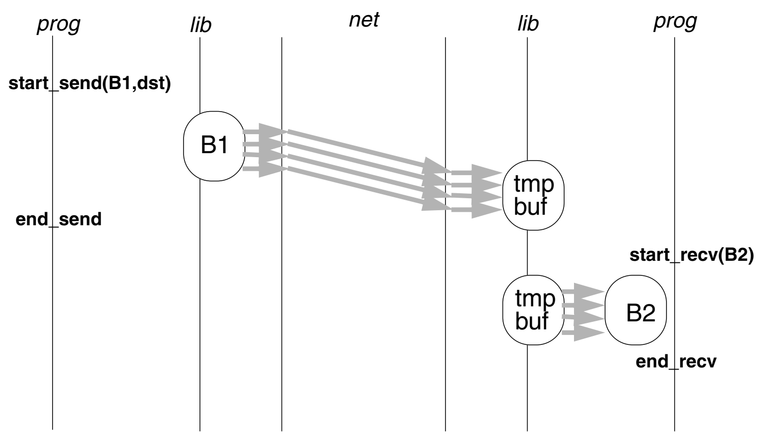


Figure 4.4: Expected communication behavior under the postal message passing model. The figure graphs the interactions between two processors on along a wall-clock time-line (top) to (bottom).

The deadlock prone example from Section 4.2.1 is no problem for a postal message passing system as seen in Figure 4.5. Regardless of the race condition the postal message passing system resolves both cases of receives. In the optimal case (arrival of the expected message as first message) it can just result into a direct data transfer like the rendezvous message. In the other case (arrival of the wrong message as first message) the buffering mechanism reorders the message to allow completion.

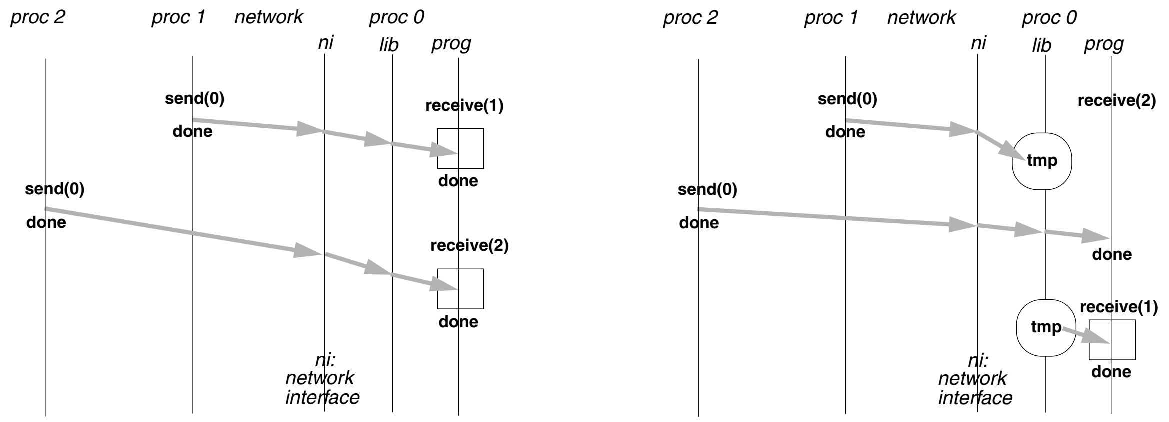


Figure 4.5: Example of a flow synchronized program that executes correctly under the assumptions of the postal model of message passing regardless of the timing and order or receive calls.

There are two common problems with implementations that are following or pretending to follow the postal model. First, there are cases of errors due to insufficient buffer space. The handling of insufficient buffer space is very tricky and portability is most likely to end once a program is operating near the buffering limit. The author knows of a parallel machine that featured an OSF/1 implementation using message passing communication for demand paging to disk space in the I/O nodes while the message passing system allocated part of its buffers in demand paged virtual memory. Such dependencies can result in the most unpredictable behavior under heavy loads. Second the performance of libraries offering the postal model is generally below par, particularly when used for compiler-generated code. Section 4.3.3 provides a more detailed discussion of the potential costs of using flow synchronized programs with postal messaging due to extra copying of the data in the buffering mechanism.

The shortcomings of the rendezvous model (difficulty to write correct programs) and the postal model (impossibility to provide a correct implementation with unlimited buffer space, and difficulty to provide an efficient implementation) lead to the definition of a new model that is well suited for efficient communication in compiled parallel code.

4.2.3 Deposit/Fetch model and well-synchronized Programs

The deposit- and the fetch- models both assume that the active part of communication is handled by one communication partner alone. The deposit model assumes the sender takes the active role and pushes the data over, while the fetch model assumes that the receiver takes the active role and pulls the data over. For a proper definition of the fetch and deposit models the specification of a receive- and a send-operation needs to be refined. The simple notion of a blocking send- or receive-call is no longer sufficient to describe the semantics of implicit sends and receives or of multiple outstanding receives and sends.

A receive operation is called *posted* or *outstanding*, if the receiver has designated some storage location for the data of a particular message. The receive is called *complete* if a matching, incoming message has arrived, and the data is placed into the designated locations. Similarly a send operation is called *posted* or *outstanding*, if the sender has designated that the data in some storage location is ready to be sent as a particular message. The send is called *complete* if a matching, incoming fetch message has arrived, and the buffer containing the data is no longer needed for communication.

Although a definition of split send- and receive-calls is crucial to define the new direct deposit model the definitions have been established in the past by several systems. Many existing message passing APIs include spilt send- and receive-operations and most standard libraries include split operations since quite a while (e.g. Intel NX or Express). Furthermore the concept of posting an implicit, collective receive for an entire class of messages by installing a user defined handler has been described as active messages [108].

The definition of fetch- and deposit- messaging relies heavily on split-phase or implicit and collective send/receive operations. The corresponding program properties for correct execution will require the distinction between data and control messages as introduced in Section 4.1. The most important point of the definition in the two models is that the "post in advance" requirement does not hold for all data messages but not the control messages.

The conditions for consistency and correctness of execution under the deposit/fetch models establishes a program property for correct execution. The programs complying with this property are called *well-synchronized* programs. There is a large amount of symmetry in the definitions for fetch- and deposit-messaging and the two models are duals using exactly the same synchronization semantics. The two cases of deposit (sender initiated transfers) and fetch (receiver initiated transfer) are distinguished solely for a precise description in the usual terms of send and receive calls.

Transfer Property 4.3 (Fetch- and deposit messaging) *The deposit and the fetch model provides no synchronization with the data transfer. The active node specifies all addressing necessary for the data transfer and the passive node executes its part without explicitly programmed participation.*

The no-synchronization-at-all semantics are obvious from the intuitive meaning of the terms deposit and fetch. With deposit and fetch alone it would remain almost impossible to provide any efficient form of consistency to a parallel program without some extra synchronization primitives. Therefore those primitives alone are not sufficient to provide a message passing model. The problem is that the passive end must always be ready - so the implicit receives of a deposit or the implicit sends of fetch must be posted in well advance. Without synchronization there is not way of telling whether this has happened or not. The fetch and deposit primitives are to be used in conjunction with conventional, blocking send/receive messaging or with dedicated synchronization primitives (e.g. hardware barriers, special memory location (semaphores) for locking).

The key idea that makes the deposit and fetch models practical and efficient, is to allow for a different treatment of control and data messages. The aforementioned ”post in advance” requirement is demanded for all data messages but it is *not required* for control messages. Control messages can be used to synchronize programs sufficiently to guarantee that each data transfer encounters a previously posted send or receive. Control messages are small and are assumed to be buffered until they are explicitly received by the processor at the receiving node or until they are handled by the corresponding protocol state machines of e.g. an implementation of a global barrier.

The derivation of a message passing model from the deposit and fetch primitives and their transfer properties raises the question of the corresponding program property for correct execution.

Program Property 4.4 (Well-synchronized programs) *A message passing program is said to be well-synchronized if the passive operation for data transfers are initiated strictly before the corresponding active operation is executed. Clearance at the passive operation can also be given implicitly. For a deposit the passive operation (clearance) at the receiver R must be given strictly before the corresponding send (deposit) operation is executed at the sender S. For fetch the passive send operation at the sending node S must be cleared strictly before the corresponding receive (fetch) operation is executed at the receiver R.*

A simple example of a well-synchronized message passing program is a parallel code whose receivers always pull the data from a sender as soon the storage location to hold a message becomes available at the receiving node. The message requesting the data transfer is small containing only control information and does not have to meet any “receive post in advance” requirements. In this case, the originator of a data transfer is the destination node, which “pulls” the data. (Depending on the hardware platform, there are many options how such pulling can be implemented.) Notice that the ability to “pull” data is not enough. Unless the application has certain properties, the receivers must be told when data is ready to be ”pulled” and additional control messages are required. Communication of well-synchronized programs that use pulling is said to be in the fetch model.

Alternatively a well-synchronized program could use pushing in a similar way resulting in the deposit model. Again the fact that the program is well-synchronized provides a predetermined destination for all data transfers and once a transfer starts the data can be deposited in its final location. The mechanics of this implementation are called *direct deposit* if the store addresses are also produced by the sender. See Section 5.2 in Chapter 5.

Figure 4.6 illustrates the expected communication behavior of a transfer under the deposit model of message passing. To make a program comply to the requirement of a well-synchronized arrange the code so that a global synchronization operation gives clearance in advance for all subsequent data transfers. Usually this synchronization step is implemented as a hardware barrier or through a software barrier built with separate control messages. After the synchronization step (barrier) all necessary assertions are established and it becomes precisely known where each data transfer of the communication step has to go. It is also established that these storage locations are ready to accept the data and the deposits of data (grey messages) can proceed without delays. Moreover non-deterministic message delivery can take place as long as the data is identified with a destination address (as this is the case with a direct deposit mechanism). There is no actual receive call invocation and the posting of the receives is implicit in the barrier. For coherency the implementation keeps track of the number of elements deposited at the receiver or the outstanding deposits at the sender so that an application knows through a semaphore or an outstanding reference counter when all deposits of a global operation have completed.

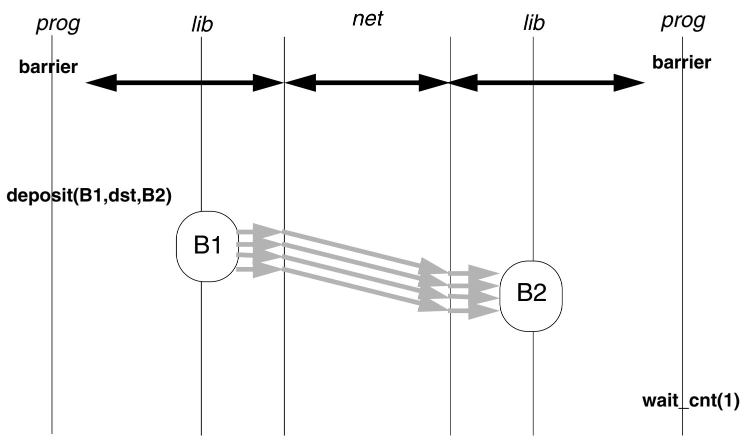


Figure 4.6: Expected communication behavior in the deposit model. The figure graphs the interactions between two processors along a wall clock time-line (top) to (bottom).

The well-synchronized message passing program requires some properties of the strongly synchronized program (rendezvous model) but also assumes some properties of the flow synchronized programs (postal model) as well. The deposit and the fetch model implementations have with the rendezvous model in common that the storage locations for all incoming data messages are fully known and specified at the time the data is exchanged, so buffering early messages can be avoided. The new model inherits its guarantee that the message passing system can deal with non-deterministic message arrival from the postal model. This last property is extremely important for dealing with realistic networks.

However to be practical, it is not enough to demand that a well-synchronized program posts the receive operation prior to the execution of the matching send operation. The passing system must allow the program to post *multiple* receive operations. Multiple outstanding receive operations are a necessary condition for a well-synchronized program to make it deal with non-deterministic message delivery. If the message passing system restricts the program to just a single posted receive operation (as is the case for the rendezvous model), the network can non-deterministically deliver an unwanted message, for which the receive operation is not (yet) outstanding, and violate the definition of a well-synchronized program. Thus the capability for implicit, collective receives or non-blocking, individual receives is a prerequisite for data transfers in well-synchronized programs.

Neither the precise notion of well-synchronized programs nor the deposit/fetch models are widely used at this time. I attribute this to the fact that it is too for the human programmer to guarantee that all necessary and sufficient properties of the deposit/fetch message passing models hold. Even in direct deposit and direct fetch, the transfer mechanisms are clearly derived from message passing computer architectures and there are no shared memory hardware coherency protocols involved to help the programmer at run time with the details of coherency. A parallelizing compiler is in a much better position to generate well-synchronized programs than a human programmer, because of the dependency analysis and good knowledge of the global communication patterns.

4.2.4 Control requirements

During the execution of the parallel program, a set of assertions must be maintained to guarantee correct execution of a parallel program. These assertions are maintained trivially in message passing systems with a simple blocking send/receive API. However maintaining the same assertions becomes much more complicated when switching to split send/receive call interfaces. The basic assertions do not depend on a specific usage model and can be written down for all models. e.g. the "source buffer" denotes both, the content of a message to be sent for message passing call or a block of data to be fetched with a remote load and the "destination buffer" buffer might just be a block of memory given as an argument to a receive call or alternatively an implicitly designated set of memory locations to deposit the data into.

There are four basic assertions:

Source buffer ready (SBR) The next version of data is ready to be transmitted.

Source transfers complete (STC) Data in the buffer has been transmitted to all requesters, so the buffer can be safely rewritten.

Destination buffer ready (DBR) Data in the buffer is no longer being used locally, so the buffer can accept new data.

Destination transfers complete (DTC) All outstanding data transfers have been complete, so the destination buffers contain valid data that can be used in a computation.

All communication models must ultimately maintain these four assertions, but various usage models can satisfy these requirements at different times and in a different partial order. A message passing mechanism becomes a message passing model only after certain relationships between these assertions are specified and constraints are imposed on these assertions. Remote stores, remote loads, and active messages are only mechanisms (i.e. the execution of a few store instructions, a few load instructions or the invocation of interrupt handler upon message arrival). A mechanism can be the basis for a message passing model once the control requirements are incorporated. Remote stores can become deposits, remote loads can become fetches, active messages can implement either of the models depending on what code is executed in the handler.

The postal model maintains these assertions largely by buffering early arrivals of messages until the matching receive is invoked. In the postal model there is no need for assertions to span multiple communication partners as long there is enough buffer space available. The assertions are established locally between the calling application code and the message passing library. In libraries with only simple, blocking send- and blocking receive calls the assertions are just the obvious pre- and postcondition for the input and the output parameters of the send and receive function calls. For libraries with split send and receive calls the assertions show up in the small print of the precise semantics of the start_send, end_send, start_receive, and end_receive calls. Figure 4.7 shows the standard order of assertions under the postal model.

While the postal model manages to get by maintaining the assertions locally, the rendezvous model forces all the assertion to become global (become known to all communication partners in a transfer). While the receive assertions are propagated by control messages (or flow control) the send assertions are propagated by the data transfer itself. Therefore after a rendezvous all four assertions hold globally. Figure 4.8 shows the standard order of assertions under the rendezvous model.

Application programs following the deposit and the fetch models must explicitly exchange control messages to satisfy the control requirements. The deposit and fetch usage models may inspire completely different programming styles, but given a parallel program, in the end, all implementations must maintain

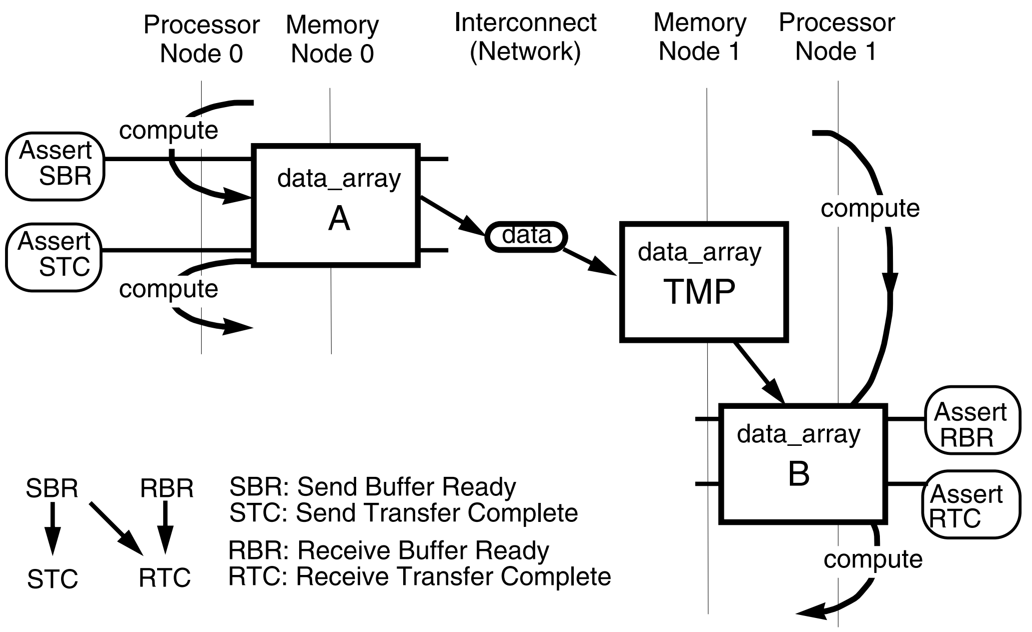


Figure 4.7: Control requirements and assertions in the of the postal model.

the same set of data transfer assertions to ensure correct operation. For example, the existence of a remote store does not solve the problem of avoiding a premature read of the remote data. A node can correctly remote load remote data only after the data are computed (and stored in the correct place).

In the fetch model, the destination controls the transfer. The destination node sends a fetch request to another node, containing the addresses of the data elements to fetch. The destination also computes the local destination addresses for the transfer. The memory system of the source node replies to fetch requests by sending the requested contents of its memory, unconditionally.

Figure 4.9 depicts the relevant steps in a fetch model data transfer. Node 1 generates the remote addresses and sends a request for these items to node 0. The memory system of node 0 replies directly with the requested data as in a shared memory load. Upon reception of the data, node 1 generates the local addresses and stores the incoming data in its final destination.

An external mechanism is necessary to tell the destination node when the data on the source node have been computed and are ready to be fetched (assertion SBR). Locally, the source node must also guarantee that all prior requests for the old data have been made before it writes new values into a buffer (assertion STC). Since the request is explicit, the other assertions are maintained automatically (as side effect of controlling when to make the request). The fetch call is not made until the destination buffer is ready (DBR), and once the fetch has completed, the new data is guaranteed to be in the buffer (DTC).

In the deposit model, the source node controls the transfer (Only one message is required to transfer the data elements together with sufficient information about their destination addresses). Using e.g. a direct deposit mechanism, the source node computes the local source addresses [1] and the (remote) destination addresses for the transfer. The memory system of the destination node then deposits the contents of the message directly into memory based on the addresses provided.

Figure 4.10 depicts the relevant steps in a deposit model data transfer. Node 0 generates the local and

[1]Computing the remote addresses may be difficult for an application programmer but can be done by a compiler, given that a good compiler has access to the details of memory layout of data.

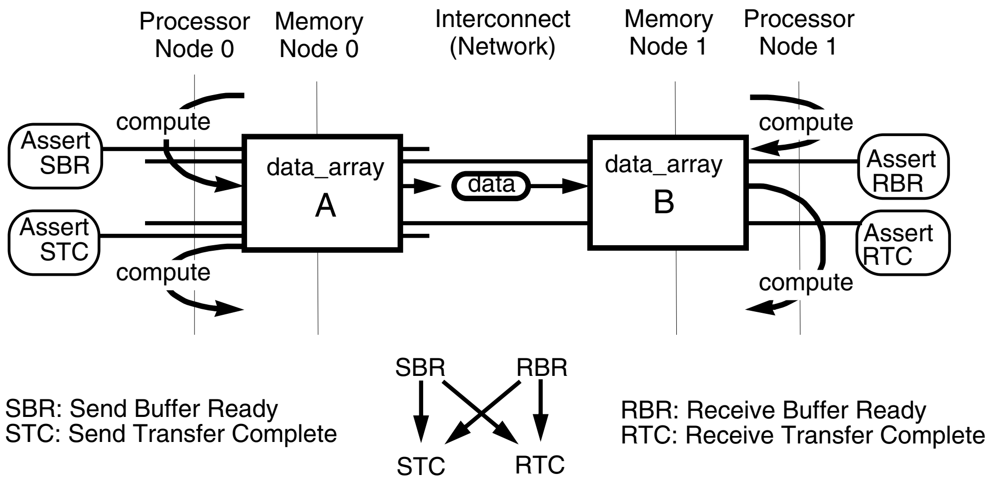


Figure 4.8: Control requirements and assertions in the of the rendezvous model.

remote addresses and sends the data together with the remote address(es). The memory system of node 1 accepts the transfer and deposits the data directly into its final destination, without further involvement or computation on node 1.

An external mechanism is necessary to tell the source node when the destination buffer is ready to accept new data (assertion DBR). Locally, the destination node must ensure that all transfers have completed before using values in the target buffer (assertion DTC). Again, the explicit deposit operation maintains the other assertions. The deposit operation is not executed until the source buffer can be redefined (STC) and has the correct data (SBR).

The fetch and deposit models can be used with several different mechanisms to maintain the four correctness assertions, unlike the postal message passing model, which must rely on internal buffering to preserve correctness. The STC and DTC assertions require only local information to guarantee that the local buffer can be used, but the SBR and DBR assertions require cooperation with a remote node to guarantee that the buffer on the remote node is in the appropriate state. This cooperation can be communicated through additional control messages, but the protocol to exchange messages while avoiding deadlock can be complicated. For machines with support for fast barrier synchronization hardware (like the Cray T3D), the barrier synchronization can act as a guard to preserve the assertions. For machines without support for fast barrier synchronization, a tree of control messages could serve a similar purpose. In some cases, e.g., if the compiler has sufficient information about the communication patterns, the control messages that guarantee the assertions may be piggy backed onto the data transfers of previous communication steps. In these cases, no additional synchronization or control messages are needed to preserve correctness.

4.3 Services of Message Passing Libraries

The first part of this section is devoted to a reference model of all necessary communication steps in message passing. The reference model is used to define a common basic terminology for the discussion of implementations in common message passing libraries. The second part of the section relates those basic steps to the actual communication services provided in most communication libraries.

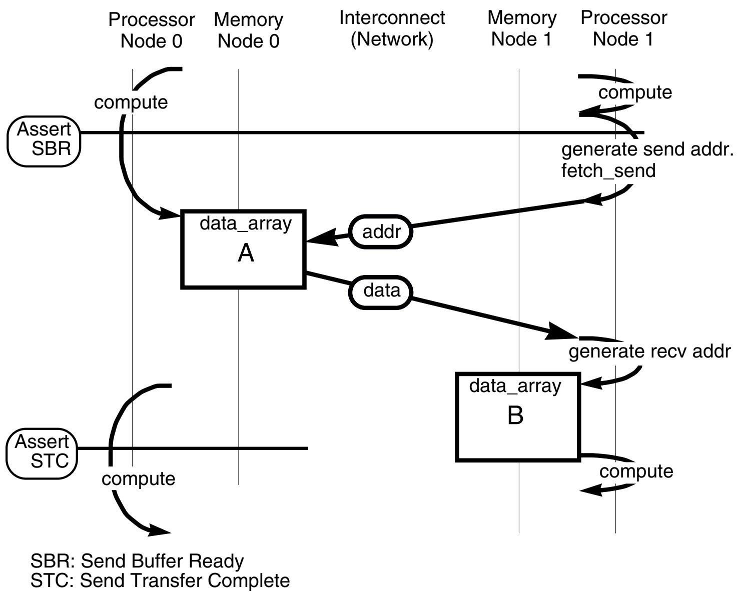


Figure 4.9: Control requirements and assertions in the of the fetch model.

4.3.1 A reference model for services provided by message passing systems

The minimal core functionality of every message passing system includes transfer of data and, along with this transfer, at least a (one-way) synchronization. Remote loads, stores and other bare-bone remote access primitives that do not provide any synchronization, but as mentioned before they are just mechanisms and can hardly be called a message passing system.

Transfer of data: As its most important function, message passing libraries copy data from the memory space of one processor to another processor within a parallel system. Different libraries provide data mover services of different quality. All libraries can handle at least contiguous blocks of data efficiently.

Synchronization: A receive operation can never complete until the corresponding send operation is invoked. This provides a one-way synchronization of the receiving process with the sending process.

However, the services provided by different standard message passing systems go much further, and not all of these services may be required for all cases. For the discussion of an efficient implementation it is necessary to identify the services that require to copy the data of a message. This list of services commonly includes:

Task management: Task management was introduced by PVM, which aimed to provide a full parallel virtual machine to the user. Task management support provides a machine independent interface to to several common operating system dependent services like initial loading executables on multiple processors, grouping processors together and spawning tasks across the network to other parallel processors at run time. While these services require interprocessor communication they are not basic communication services.

Reliable transport: A service that is generally taken for granted in a parallel computer is **reliable data transport**. Unlike in network computing, the interconnect of a parallel supercomputer can either

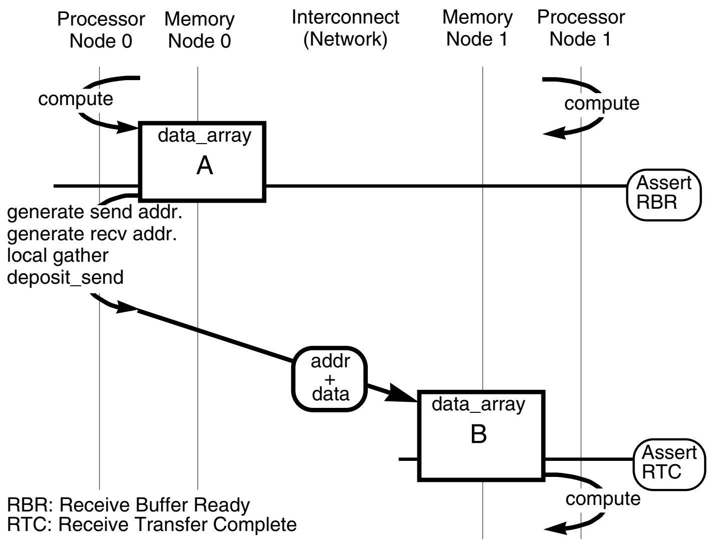


Figure 4.10: Control requirements and assertions in the of the deposit model.

correct or flag all transmission errors, and the rare case of a transmission error is considered a machine failure, and the computation is recovered by restarting from a checkpoint. The first generation of network computers rely on interconnect technology without strict resource and bandwidth allocation and can therefore not give the same guarantees. Message passing libraries could provide timeout and retransmit although this is often left to a separate underlying software layer such as TCP. Reliable transport on an unreliable network requires a logical copy of the data [2].

Packetization for the network: A service that is required to improve load balancing in the network and deal with networking standards that prohibit long messages. The service is performed in hardware in many supercomputers and in the second generation of high speed network adapters. Although there are copies involved, they are done by dedicated hardware and usually are not a performance bottleneck.

Protection: Operating and runtime system messages must be decoupled and protected from user data messages. In some machines, messages from different users must be protected from each other. Even in space shared machines it is important to prevent the accidental transmission of messages to non-participating processors or tasks and prevent an accidental consumption of messages addresses to the OS or to other users. Most network interfaced require that all accesses to them are protected in kernel memory space and that they are serviced by privileged code. In this case a copy or a remapping of pages is unavoidable. Newer network adapters allow to deliver data directly to user space on a per connection basis [107].

Buffer management: Some message passing systems accept the data in a simple blocking send primitive but move it into a system buffer before injecting it into a network. Other systems extract data from the network into system buffers and hand the data over to the user only after the receive primitive is invoked. Message passing libraries with buffering often manage the buffers to help the programmer with storage allocation and buffer management.

[2]Under certain condition a replication of the data elements can be avoided by remapping memory pages.

End-to-end flow control: The basic one-way synchronization service of a message passing library can be extended so that the receiver can also synchronize the sender by forcing the sender to slow down (or even stop) sending. This service is implemented as end-to-end flow control and independent from the link-level flow control usually found in the network hardware. Most networks are operated under the assumption that link flow control just regulates smooth flow between intermediate nodes and is not used as end-to-end flow control to back up data on specific connections.

Selective receives: The programmer needs to match corresponding send and receive operations to assure proper handling of the data transfers. On systems where multiple sends and receives can be outstanding, the message passing library must have a mechanism to select the proper message for each receive operation.

Transferring data is the basic and most important service of a message passing system. A pair of send- and receive-calls provides this service, regardless of the programming model. The major difference between message passing models and many subtle differences in the implementations involve the type of synchronization service that is also provided. The synchronization service is required whenever the sender and the receiver invoke a data transfer at different times. Such a synchronization is provided either involuntarily or by choice, when different services are integrated into the same message passing primitives.

Most services of message passing libraries are orthogonal to the synchronization model. Whether a library offers protection depends on the operating system and user environment (dedicated supercomputer vs. shared cluster of workstation). The needs for packetization and reliable transport are mostly determined by the underlying network architecture. The question of buffer management and selective receive is mostly determined by the API of the library. Task management is completely optional and could be done equally well outside the message passing library. However there is the important service of buffering that is directly linked to the synchronization semantics.

4.3.2 Interaction between synchronization model and buffering

All postal and rendezvous model message passing systems provide some implicit synchronization service with each data message that is transferred between the sending and the receiving process. It is either one-way or two-way depending on the underlying model. The most relevant difference between these models is the minimal and maximal amount of synchronization required and permitted by the model as side effect of a data transfer. Both models provide a minimal one-way synchronization that requires data to be produced and sent before the receives complete. If there is no data, the receive operation blocks and waits for the data. The maximal amount of synchronization differs. While a two way synchronization is permitted or even required by the rendezvous model, it is strictly forbidden by the postal model.

Theorem 1 *One way synchronization semantics (the unrestricted postal model) implies the need for a buffering mechanism with unlimited buffers in its implementation.*

The truth of this theorem becomes obvious when inspecting the informal definitions of the semantics of the send statement in the postal model. In the postal model sends required to be "fire and forget", i.e., they are accepted and handled by the message passing system independent of the matching receive. Therefore some buffer storage space is necessary to allow a send operation to complete before the corresponding receive is invoked. If a transfer would run out of buffers, it would back up the data and accidentally provide a two way synchronization violating the postal model.

More precisely the semantics of a send operations in the postal-model require that the data is accepted by the message passing system eventually and unconditionally. As soon as the data is accepted, the data structures holding the data must be released. More precisely a message passing system can delay the completion of a send call for some time because of network congestion and still implement the postal model, but this blocking and delaying can neither directly nor indirectly depend on the receiving side of transfer (Split send calls do not solve the problem of copy semantics and are merely a mechanism to put buffer management under user control - the extension of the deposit concept to split calls and compiler managed buffers is to be discussed in Section 5.6.1 of Chapter 5).

This "fire and forget" property of the postal send implies that sufficient buffer-space must provided somewhere along the route to hold messages that are sent before a corresponding receive is posted. Depending on the link level flow control protocols used, buffering can be provided at the sending node, en-route (virtual cut through), or at the receiving node. In most libraries buffering is implemented at the receiving node. Some libraries such as Cray PVM 3.0 and NX rely on a "pull" mechanisms for long messages and provide bufferspace at the sending node because the sender knows the size of the buffers.

A message passing system that just blindly dispatches messages into the network without buffering (e.g., into a wormhole routing network) could cause a situation where the addressed receiver is not yet ready to extract the messages. The messages in transit might be long enough to fill the entire path in network and delay the sender until the receiver starts processing the message. All supercomputer backplanes and many newer ATM networks provide link level flow control mechanisms to guarantee reliable transmission between two endpoints. However in most practical networks, some unrelated messages share communication channels and potentially block each other through flow control mechanisms like the ones described in Section 2.2 of Chapter 2. It is well known that supercomputing networks are only deadlock-free under the assumption that all receivers continuously and unconditionally remove incoming messages [28].

This unwanted mutual blocking of unrelated messages makes buffering a requirement to stay clear of resource deadlock within the network. Conceptionally buffering works as long as the storage space is not bounded. If storage space is bounded (as it is in practice) some flow control between sender and receiver is required to prevent buffers from being overrun. Deadlocks are immanent unless the programmer adapts to the changes in the message passing model due to these limitations. For well engineered parallel programs the maximal amount of buffer space is certainly bounded, but it proved to be harder to determine the maximal amount of buffers than to determine the maximal running time. The usage of buffers can depend on the relative execution time-lines of the different processors and on the skew between processors. Therefore the practice to deal with the limited buffers of message passing libraries is handled more like an art of voodoo than a problem of computer science. PVM and MPI have load-time parameters for internal buffer-space, and programmers regularly tune these by trial and error.

A similar question is raised by the rendezvous model. How much buffering is required or allowed for a correct implementation of that model.

Theorem 2 *A two-way synchronization semantic (the basic rendezvous model) restricts the amount of buffering to the minimal unit of transfer or less.*

Lets assume that there is more buffering space than there are elements in the message to be sent. In this case the two-way synchronization semantics of a data transfer might get lost and the assumption of the rendezvous model is violated. This confirms the theorem. The theorem does not limit the number of buffer positions available along the communication but the number of position usable along the path. If extra buffer space is available, the protocol messages (request, reply, acknowledgment) in Figure 4.1 are used to prevent the use of that buffer space and enforce proper two-way synchronization. This restriction is a problem in the implementation of axiomatic, simple parallel languages like OCCAM, where the basic unit of a message is just a word. The maximal message size of a single element and the preclusion of any pipelining limits the performance of communication on any practical high speed network.

Limitations on pre-allocated buffer space mandate that long messages are broken down into smaller packets, and that additional control messages are used to send requests for buffers, replies, and acknowledgments in messages. Flow control protocols provide unwanted synchronization (beyond buffer management) and doing so is expensive. Each data transfer has implications on the management of storage, and this coupling increases the cost of the data transfers, since each transfer may involve storage management decisions.

4.3.3 Reference model of communication steps

Figure 4.11 illustrates the different steps of traditional message passing implementations as they are performed for a typical parallel computation.

For **the sender side:**

S.T (Send Transfer): The data of a message is transferred from user space into system space. The send function may complete at this time, and the user may or may not be free to overwrite the data.

S.K (Send pacKetize for network): Message wrappers and routing headers are generated. This step potentially breaks large messages into smaller packets. All outgoing message packets are queued.

S.I (Send Inject): If the network is ready to accept another message, it is injected into the network.

and correspondingly for **the receiver side**:

R.X (Receive eXtract): If a message is available, it is extracted from the network.

R.D (Receive Depacketize from network): Packets are reassembled into messages if necessary. Incoming messages are buffered or matched with pending receives.

R.T (Receive Transfer): The data is transferred from system space to user space and the internal buffers are released.

From a compiler's point of view, Figure 4.11 is overly simplistic, since it assumes that parallel programs always exchange dense and contiguous blocks of data. In practice, the data to be exchanged are usually stored in (fields or elements of) program variables, and they are not generally stored in contiguous memory locations. Therefore, the basic steps must include two more operations:

S.P (Send Pack or local gather): The out-bound data is gathered from its original location and put into blocks for more efficient transfers.

R.U (Receive Unpack or local scatter): The in-bound data is stored in its final location.

Both operations include the corresponding address computations. Figure 4.12 augments Figure 4.11 with these operations.

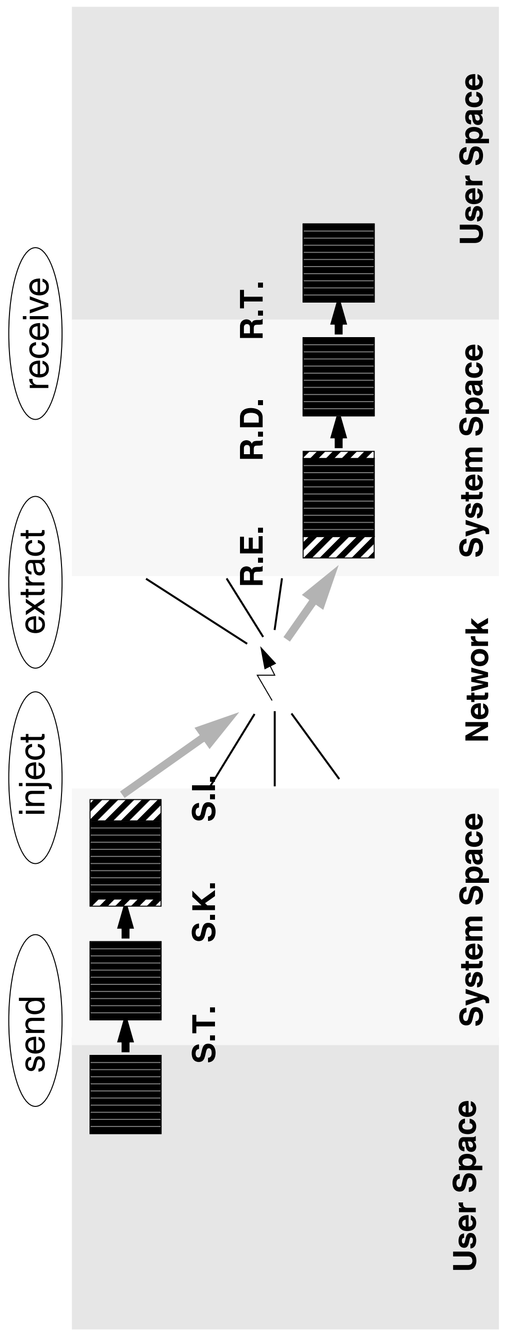


Figure 3.11: Reference model of basic communication steps for traditional message passing libraries.

Figure 4.11: Reference model of basic communication steps for traditional message passing libraries.

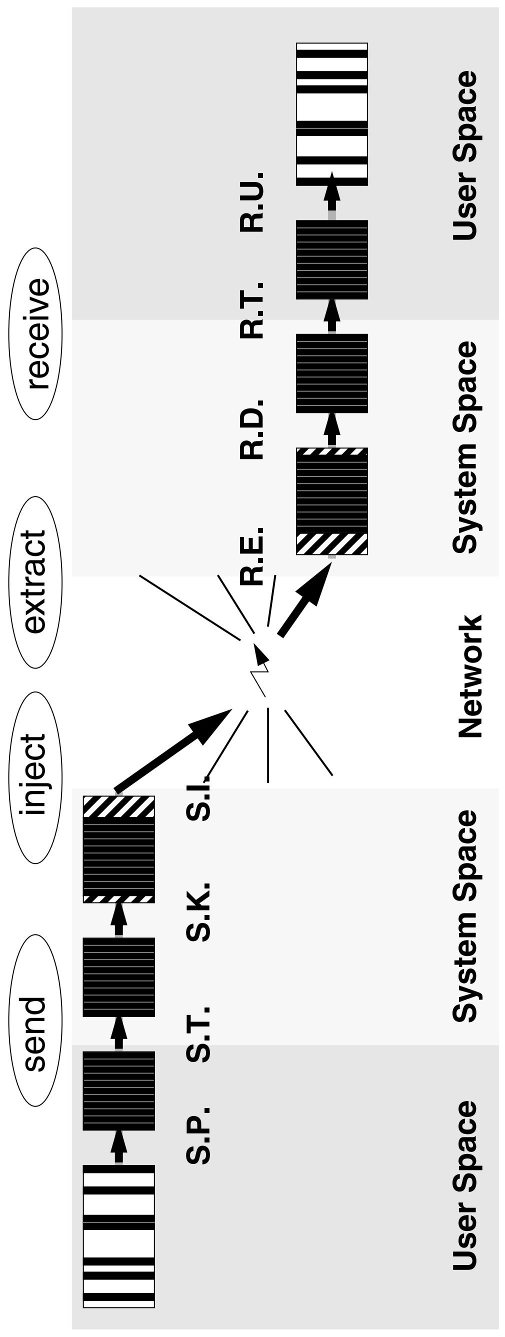


Figure 3.12: Reference model of basic communication steps for traditional message passing libraries - including extra steps within the application for packing and unpacking of the message data at both end-points.

Figure 4.12: Reference model of basic communication steps for traditional message passing libraries — including extra steps within the application for packing and unpacking of the message data at both end-points.

4.4 Synchronization models vs. cache coherency models

It is interesting to point out the similarity of synchronization semantics in message passing to the different coherency models in shared memory systems. The introduction of weak consistency for shared memory parallel computer for the sake of better communication efficiency and the implementation of several prototypes have triggered an extensive investigation of research in the area of shared memory consistency. The results were better models [2], the design of better cache coherency protocols [4] and even novel approaches to the verification of shared memory hardware [73], just to name a few of many recent contributions in that area. In a sharp contrast to that, there is little interest on the corresponding problems for message passing systems. Little work has been done to overcome the difficulties to capture the detailed synchronization semantics and to come up with precise a definition of message passing services. One exception is probably an attempt to define an operational semantic for MPI [27], which is extremely cumbersome and almost intractable due to the rich features of a full standard library. The compiler writers and the parallel application programmers still struggle with the problem of subtle limitations and undocumented features of their message passing libraries.

Chapter 5

Decoupling Synchronization and Data Transfers

Based on an examination of different message passing models in Chapter 4, we describe the different roles of control and data messages and define well-synchronized programs. Well-synchronized programs meet the criteria for correct execution in the deposit model of message passing. The model postulates a particular relationship between control and data messages by requiring that all data transfers are pre-synchronized with some control message in advance. In this chapter we argue such decoupling of synchronization and data transfers simplifies the implementation of message passing libraries and enables a number of optimizations that drastically increase the efficiency of communication code. We introduce the "deposit" and the "direct deposit" message passing mechanisms and show how decoupling synchronization and data transfers is crucial to make these two mechanisms work correctly and efficiently. The idea of decoupled synchronization and data transfers is independent of deposit messaging and leads to optimization applicable even to programs using standard message passing libraries like PVM [102] or MPI [42]. Separate synchronization and the deposit model are often tacitly assumed in newly proposed fast message passing mechanisms like active messages [108].

5.1 Decoupling increases performance

Separation of synchronization and data transfer is the key to communication performance in parallel supercomputers. Fully decoupled and independent synchronization generates many opportunities for improvement and optimization of both the synchronization and the data transfer component of a communication system. Optimizations include simple and fast communication hardware as well as simple and well-structured system software. The biggest difference is caused by delegating buffer management to the application. Global information about the communication pattern in compiler-generated parallel programs makes it easy to manage the buffers involved. Based on the same knowledge the compiler inserts additional synchronization primitives until the generated code qualifies as a well-synchronized program. Once the program is well-synchronized its data transfers no longer require buffering and the buffering mechanism are eliminated. Decoupled compiled parallel programs do not depend on individual protocols to perform data transfers. It is no longer necessary to request and acknowledge the availability of each individual buffer allocation; requests and acknowledgments can be provided for a whole buffer *pool* rather than for an individual buffer. Pools are managed as a unit by the compiler.

There are four ways in which the separation between control and data transfers can simplify message passing systems and improve performance. Decoupled message passing uses this separation to:

Target data to the final destination Synchronization is required prior to data transfers, if we want to target the data directly to its final destination (which is always desirable to avoid copying).

Put buffers under user control . In communication systems without system buffers, all data transfers are fully under user control (desirable to avoid copying). Therefore we cannot rely on any synchronization provided by data transfers. A separate synchronization mechanism and/or a global synchronization concept is required.

Eliminate the need for buffering . If synchronization and data transfers are coupled, a combined control and data message may involve buffering and costly storage management operations, since these are necessary for data transfers to complete. With prior synchronization we can be sure that there is no buffering and eliminate all the housekeeping overhead for buffering in the message passing system.

Combine control messages into global communication primitives . If synchronization and data transfers are decoupled, control-only messages can be combined into cheaper global operations like hardware barriers. This algorithmic optimization reduces the message complexity of many flow control protocols from $O(P)$ to $O(log(P))$. P is the number of processors.

Message passing models with decoupled synchronization and data transfers simplify the design of message passing libraries and permit the exploration of novel mechanisms for data transfers and new communication optimizations. Better data transfer mechanisms together with aggressive global optimizations are the basis of the new Catacomb communication code generator used to generate communication in the Fx compiler back-end [95, 94].

5.2 Mechanisms to move data

Different message passing libraries employ a variety of mechanisms to transfer data. Some libraries push data into the receiver; others push a short request and the pull the long data block. Some libraries poll for messages while others use interrupts to signal message arrivals or the willingness of the network to accept more data.

Along with the development of the deposit and fetch synchronization models we investigated corresponding data transfer mechanisms that work well under this model. The term deposit message passing system (DMSG) refers to a particular message passing mechanism that relies on decoupled handling of control and data transfers. Similarly the fetch mechanisms can fetch data blindly from the source location in the memory system of a remote processor. Both mechanisms require well-synchronized programs to execute correctly. The mechanisms can work in that way because a separate synchronization operation has obtained clearance for that data transfer in advance, as specified by the deposit or fetch model. In the former case the data messages can be deposited into local memory at the final destination immediately upon arrival; in the latter case the addressed data elements can be fetched from the remote memory of the processors, targeted by the fetch.

The notion of coalescing multiple elements into a message for a deposit transfer contrasts the deposit mechanism with non-cache-coherent remote loads and stores. A single deposit message can scatter data elements all over the local memory of a destination processor but still the source and destination processors are set up for a large transfer and any per message-overhead is paid only once. Since this notion of a message is preserved, the hardware or the compiler knows that a series of remote accesses is performed and can use deep pipelines for copy loops or exploit any architectural support a machine might provide for such a case. Similarly, a fetch is viewed in a highly pipelined fashion. A fetch transfer is initiated by a message consisting of addresses. The destination node then responds immediately with a message full of data elements. On machines with perfect hardware support for remote loads and remote stores, fetches and deposits might be quite simple. From our experiences with the T3D and the T3E, we know that on both machines a few auxiliary registers must be set and the copy transfer loops must be software pipelined accordingly to achieve maximal bandwidth. We think that the difference in this per-message overhead justifies the distinction between deposit and remote store.

5.2.1 Direct deposit messaging

In the *direct deposit* messaging system with decoupled synchronization, all messages are taken directly from memory (user space) at the sender and are automatically directed to their final destination in the memory at the receiving end. Direct deposit resolves addressing at a fine granularity and permits transfers of a collection of data elements directly from their source locations to their final locations. If temporary data structures or buffers are used, they are under compiler/user control and synchronization messages are generated separately. Like the receive-less deposit mechanism, direct deposit dumps its data blindly and cannot establish any synchronization properties with its data transfers. There exists a possibility of live data being overwritten and the receive mechanisms can not address the problem. A decoupled form of synchronization is needed for correct execution and for data consistency across the processing nodes.

The synchronization model does not determine how the address information is specified for the transfer; it merely specifies that the final address for the transferred data is known at the time of the actual data transfer across the network. Multiple solutions to the problem of address translation remain possible. We qualify the deposit messaging system further according to the way the addresses are specified. In *direct deposit messaging* all addressing information is sent together with the data elements; in *indirect deposit messaging* the addressing information is held available in a table at the receiver node. These are only two out of many possible options to specify the addresses for transfers in deposit messaging. An active message handler could calculate the addresses on the fly upon message arrival using the receiving processor. A further dimension in the space of addressing modes is whether the addresses associated with the data are physical, global virtual, or local virtual. The default strategy of memory allocation on most supercomputers remains still physical memory allocated in large segments without paging. Most advance network adapters (e.g. fast ATM adapters) that allow direct deposit require that all memory pages involved in this process are pinned into physical memory beforehand (e.g, HARP [77]).

5.3 Optimizations in decoupled messaging

Decoupling permits a series of global communication optimizations to increase the performance of communication code. Some of the optimization are short cuts to the simpler decoupled communication while others are true global optimizations applicable when a compiler has a global view and a more detailed understanding of communication and memory access patterns than a human programmer.

5.3.1 Elimination of copies while buffering

The data transfer executed by of a general purpose message passing library, (e.g., PVM or MPI) can contain up to six copies of the data elements involved. The copies range from coalescing of elements into contiguous buffers to copies from user to system space and buffering copies required by lack of synchronization — up to three copies at the sender and three at the receiver. The steps and copies are outlined in Figure 5.1 and labeled with the terms introduced earlier in the reference model. For part of the discussion in this section all buffers are placed in user space. In this case the number of potential copies is reduced to four. There are no copies due to protection domain crossings, since on supercomputers with space sharing, a partition of nodes is dedicated to a single user.

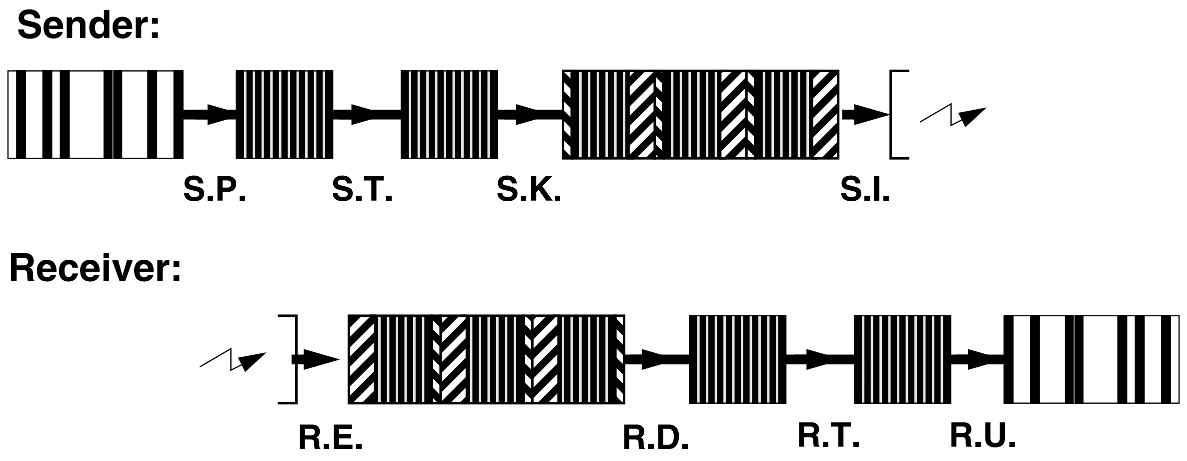


Figure 5.1: Potential copies of data elements as they occur in traditional message passing (e.g. PVM) due to coalescing of non-contiguous data transfers, protection domain crossing, and buffering to comply with synchronization semantics of postal model.

Standard message passing library implementations require buffering steps at either the sender or the receiver node to maintain the semantics of the postal message passing model because of the synchronization semantics implied by every data transfer. This explains one set of copies. The second set of coalescing copies is required since the built-in buffer management of standard libraries such as PVM can only handle contiguous blocks of data. Two more copy steps are used to gather the strided or indexed data elements out of the array structures at the sender and scatter the delivered data elements to their final destination data structure at the receiver.

Eliminating copies is the most important optimization performed in direct deposit message passing. On parallel supercomputers, interconnection networks can transfer data at rates close to the local memory bandwidth. Buffering at the end points through copying is therefore a limiting factor to communication performance, since the traffic to and from data buffers traverses the memory bus multiple times. A quantitative model relating memory system performance to message passing communication performance is given in Chapter 6 and an experimental evaluation is presented in Chapter 9.

5.3.2 Simplified buffer management in compiled parallel codes

One of the fundamental assumptions in the design of a standard postal, user-level message passing library is that the receiving node need not know at which point in time a particular incoming message will arrive or who will be identified as the sender of the next message. At any moment a message of arbitrary size could arrive from any sender; if the corresponding receive has not yet been posted, the message needs to be taken immediately off the communication network, and eventually depacketized and stored somewhere in memory (R.E and R.D). When the receive is finally posted, the message passing system must again copy the data into the user receive buffer (R.T). Finally, the user routine must move data from the user buffer into its final destination in memory, with a re-map or copy if access patterns make this necessary (R.U). Altogether we could see the message being copied on the receiver a total of 6 times. Clearly this is the most important performance problem in such a library and the best starting point for any attempt to reduce overhead. A highly effective technique for reduction of copies is to deal away with user and system protection domains (eliminate S.T, R.T) and to put buffering fully under user control.

A parallelizing compiler, such as Fx, has more information about the communication pattern and the message traffic than a general-purpose message passing system could have. Fx generates communication based on aggregate data movement patterns resulting from *array assignment statements*. Consider the array assignment statement $A = B$, where A and B are distributed arrays. The communication pattern can be complicated by two factors: arrays can have an arbitrary number of dimensions, and the array subscripts can be an arbitrary combination of triplets (e.g., $A(1 : n : 2)$) and scalars. The compiler can work together with the low-level communication system to optimize the data movement in the following ways:

Buffer allocation: In general, the compiler knows the sizes of the messages in advance, both on the sender and on the receiver. Often the exact message size can be computed at compile time, when array sizes are known; if not, the compiler can generate an expression for a fairly tight upper bound that can be calculated at run time. As a result, the compiler can allocate the buffers needed by the receiver, eliminating the need for the R.T copying step. Furthermore, the message passing system can depacketize (R.D) directly into the compiler-generated buffer, eliminating the need for that copying phase as well.

Deposit model: Separate synchronization can be inserted into a program to ensure that all receives are posted in advance, either as implicit receives (direct deposit) or explicit receives (split calls). Posting all receives in advance allows the sender or the receiver to specify user buffers ahead of the time in all cases. There is no buffering resulting from a "not yet known" destination of the incoming message, and the contents of every message can be deposited into the final data structure.

Direct deposit mechanism: When using the direct deposit mechanism of communication, receives can be made even more efficient. Because destination store addresses are sent along with the data, buffering space no longer needs to be contiguous and the unpacking phase (R.U) can be integrated into the extraction or depacketizing phase (R.E/R.D); the low-level message passing service copies data directly into its final destination in memory. When using direct deposit, the compiler does not need to allocate additional receive buffers on the receiver. Instead, in the assignment statement $A = B$, the destination array A is implicitly used as the receive buffer.

Use of the direct deposit model puts little additional hardship on the compiler. The sender already has to compute the addresses of array elements to send; a simple linear transformation gives the offset into the destination array on the receiver. Furthermore, the compiler has the ability to easily determine whether address-data pairs are necessary, or whether the address information can be compressed into address-data blocks [32].

Sender packetizing: We can also eliminate extra copying and buffer allocation on the sender side by integrating the low-level communication routines into the compiler. The compiler has the ability of directly generating the low-level packets that are sent over the communication network; thus the S.C, S.T, and S.P phases can be combined into a single phase in which the compiler generates a single packet that the communication system sends out.

The study of complex buffer management in conventional libraries is the major motivator for formulation of the deposit model and the development of the much simpler direct deposit message passing mechanism. The simplicity of the deposit approach implies that such a system has the potential to be faster than a conventional message passing system and that more hardware and software optimizations can be applied to the communication system itself and the application code written for the deposit model. From prototype implementations and code examples in Chapter 9, we infer that an HPF-style Fx Fortran compiler can use a simple model successfully and can handle the burden of additional duties like communication pattern analysis, synchronization code generation, and management of communication along with the management of its own compiler temporaries.

5.3.3 Combining protocol messages

With the global knowledge of the communication pattern a compiler can reduce the cost of the control transfer significantly, in comparison to having no global knowledge. Protocol messages of several nodes can be combined, resulting in a drastic reduction in the total number of protocol messages sent in the system (e.g., for buffer management and flow control). The most common patterns for transposes are complete exchanges (or personalized all-to-all communication) among n nodes. Such a communication step results in n^2 data transfers, one for each sender-receiver pair. A connection-oriented request/reply flow control protocol must send $O(n^2)$ messages to request and acknowledge the data transfers between all senders and receivers. However, since the compiler has global knowledge about the communication step, it can use a simple tree reduction to propagate the acknowledgments to all processors with a total of just $2n$ messages in $2\log n$ steps. The two upper curves in Figure 5.4 in Section 5.5.2 show the different $O(f(P))$ message complexities and quantify the benefit of combining synchronization messages on the T3D and the iWarp; at this point we refer to the shape of the curves only and delay a detailed discussion of the measured latency numbers until a more detailed explanation of the costs in Section 5.5.2.

5.3.4 Specializing control messages - barriers

The communication patterns for control messages are often highly regular and known to the compiler, often even at compile time. With this knowledge each node can allocate all communication resources (e.g., buffers and table entries) statically. Resources are needed only at the source and destination points of the control messages, because control messages convey only sequencing information. It is sufficient to identify them as control messages and the pay load fields of such messages can be left empty. Specialized hardware for short messages and message combining exists in the networks and network interfaces of many supercomputers. Such hardware support can perform "and/or" reductions of control messages along the communication path or route them along dedicated synchronization trees. Therefore dedicated hardware outperforms the general messaging mechanism for propagating synchronization and flow control information among the computation nodes.

The drastic performance improvement comes as no surprise. Traditionally, the general purpose message passing hardware is designed to transfer *large amounts* of data efficiently. Control and synchronization messages have different characteristics (small size, immediate use at receiver) and should be handled by different mechanisms; e.g., barrier synchronization hardware or hardwired combining trees based on state machines located directly in the network interface.

5.4 Managing decoupled synchronization

In the first part of this chapter, we suggest that decoupled synchronization can boost communication performance in a messaging system. We investigate the simplifications and name several possible optimizations, if control and data transfers are properly separated. As a target for those new messaging systems we envision a parallelizing compiler. In this section we introduce briefly how decoupled messaging (our direct deposit message passing system) can be integrated into a parallelizing compiler and how the model matches data parallel HPF application. In the light of optimized synchronization we discuss briefly the costs and benefits of globally optimized synchronization.

5.5 Inserting synchronization into compiled parallel code

The data parallel model used in many data parallel languages facilitates the insertion of synchronization into compiler parallel program. In most applications we find natural synchronization points at the border of array assignment statements, re-distributions, or explicit local transformations such as transposes. Several authors of previous work have analyzed the problem and specified how to derive communication patterns and synchronization points from either application characteristics [37] or communication dependence analysis [53].

Data intensive, scientific application codes follow a high level algorithm to solve a computationally simple problem on large data sets and therefore decompose naturally into computation phases and communication phases. Figure 5.3 shows a high level block structure of two codes in which transposes are heavily used. Two computation steps alternate with two communication steps in each iteration of the filter or the solver.

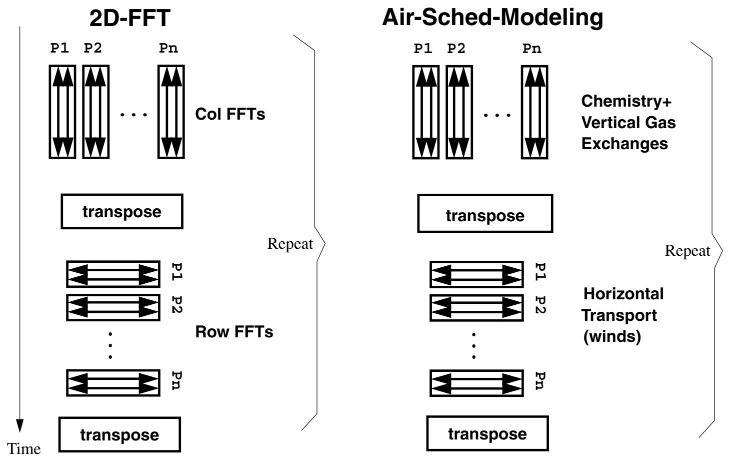


Figure 5.2: Block diagram of two application kernels showing their natural decomposition into compute and communication phases. 2D-FFT (left) and Air-shed modeling (right).

5.5.1 Phases in compiled parallel programs

The data parallel programming model of array assignment statements used in HPF exposes the common structure of the scientific application to the computation and communication compiler. This is most visible in the time-line of the generic example code in Figure 5.3. The code sketch is structured after an image processing code such as a multi-base line stereo code [111] or an image feature recognition code.

As its first two steps the application kernel requires a transformation of a 2D-image from time-domain into frequency-domain. A local filtering step is followed by an intensity adjustment using a globally calculated norm. The first part of the 2D-FFT is a simple 1D-FFT applied to all rows in parallel. To extend the

Fourier transformation to both dimensions a global communication step is required. The communication step re-arranges locality of reference for the following column FFTs and allows a local execution of the computation of the FFTs. Once the image is in the frequency-domain, a local filtering operation is applied to every data point in the image. Often this filtering operation is separated into two steps by a scaling factor computed from a global scan reduction (computation of some sort of norm). Therefore a tree like scan operation follows the local norm computation before the scaling computation continues as a next local computation step. The filtering step is followed by a reverse transformation back into the time- domain.

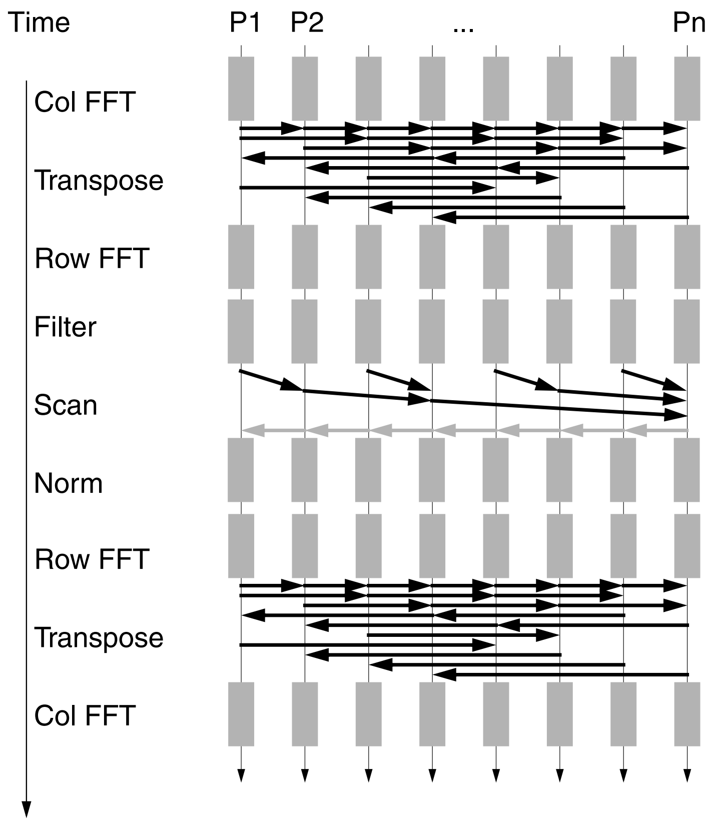


Figure 5.3: Computation and communications phases in typical applications.

Compute phase

Most application codes rely on some computational kernels derived from sequential codes or standard libraries. In Fx Fortran the `PDO` construct is used to invoke such local computation on all the data elements owned by the corresponding processor. The port of a few key application programs into Fx Fortran illustrates the dependence on existing sequential code. Even a simple 2D-FFT application kernel is not competitive without the cache optimized 1D-FFT subroutine, linked in from the machine vendors' hand tuned scientific library. A second code, modeling air-pollution in the Los Angeles air-shed depends heavily on existing sequential FORTRAN code to compute the chemical interactions between gases for each grid point and for computing the transport phenomena among grid points but separately for all chemical species and for different vertical grid-points. In a similar way parallel solvers (such as the regular SOR and the irregular FEM) typically exchange overlapping regions of data after a long local computation step.

Communication phase

The computation in most scientific parallel programs (except for a few embarrassingly parallel ones) requires a fair amount of communication between the data elements in distributed data structures [83]. In most codes such communication takes place when the data is rearranged to provide locality of reference for a subsequent local computation step or solvers exchange overlapping regions of partitioned data structures. The partitioning of the data structures involved in such communication is usually well known to the compiler, and the communication pattern is quite characteristic. We found three dominant patterns in such communication step: (1) Transposes resulting in dense communication patterns and involving nearly all data elements of a distributed array, (2) exchanges of overlapped regions resulting in next neighbor- or sparse communication patterns and involving a fraction of the distributed arrays, and (3) scan reduction resulting in tree like communication patterns involving only a few global values that are combined and broadcast during the communication step. Although the three types of patterns covered all performance critical cases of the application studied, they are by no means an exhaustive list of communication patterns. The Fx communication back-end also generates broadcasts and some global gather/scatter patterns for array initialization and parallel I/O operations.

5.5.2 Cost of separate synchronization

Decoupling synchronization raises the question of costs for separate synchronization service, as it is no longer provided by the standard messages used for control and data transfers. Will two separate control and data transfer mechanisms be cheaper than an integrated solution even if decoupling requires that additional messages are introduced?

The answer to that question depends on the protocol used to provide the necessary synchronization function. Figure 5.4 shows scalability of the synchronization costs for different implementation options in an all-to-all communication on a T3D and an iWarp. The symbols represent measured data; the lines connect these points by interpolation to show the trend. Three different ways to propagate synchronization information among all communication nodes are considered. In the first method (ctrl-msgs), each data transfer is accompanied by a request, a reply, and an acknowledge control message, representing the integrated solution and thus ensuring that the buffers can be managed easily. In the second method (ctrl-msg-tree), the same requests, replies, and acknowledgments are carried out in a collective communication operation, using combining trees. The third method (hw-barrier) invokes the subset barrier hardware of the machine. On the T3D the dedicated FAN-IN/FAN-OUT tree synchronization network is used. On the iWarp two hardware mechanisms are available to provide synchronization. The speed curve of the hardware barrier based on the diagnostic network (hw-barrier-1) shows the performance characteristics of a perfectly scalable solution at a constant, fixed overhead regardless of the number of processors involved. The constant however is high since this barrier is running at a fraction of the communication clock speed to accommodate a single long broadcast wire with its excessive inductance and capacitance and a bit serial multiplexing scheme for multiple barriers. The second speed curve (hw-barrier-2) shows a preferable barrier implementation based on a few reserved, dedicated pathways within the high performance communication system. The links and switches of iWarp permit flit/word multiplexing of up to 20 channels per link and make it easy to set aside some of those communication resources for synchronization and control transfers.

These measurements of control message exchanges illuminate the importance of paying attention to the cost of control messages. Exchanging $O(\log n)$ messages instead of $O(n^2)$ results in significantly

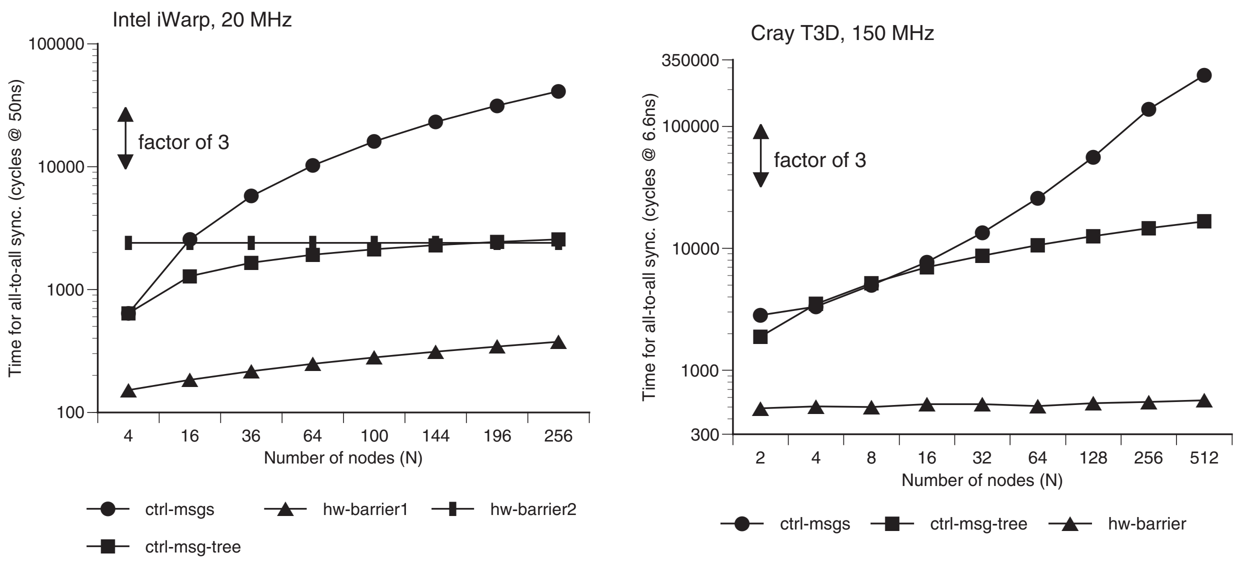


Figure 5.4: Costs of different mechanisms for control transfer (e.g. flow control) in an all-to-all communication step on iWarp and Cray T3D.

improved performance, as expected. On the T3D, both the ctrl-msg and ctrl-msg-tree implementations use the same ¡shmem_put¿ remote store mechanism on the T3D with less than one microsecond latency and overhead for one transfer. Thus the improvement is of purely algorithmic nature by using a tree instead of a fully connected graph. The further improvement for hw-barrier is due to the dedicated barrier synchronization hardware. The results show that in the control/synchronization hardware there is an optimized path for zero length messages resulting in a worthwhile option to handle control messages. The same holds for the ctrl-msg and ctrl-msg tree on the iWarp, where a basic wormhole router messaging system is used. The further improvement for hardware barriers is again due to dedicated hardware (a set of hardwired pathways) and short-cutting of message handler overheads for single word messages with zero length payload.

Unless mentioned otherwise, our DMSG deposit messaging system relies on hardware barriers for synchronization. For a separate evaluation of control transfer optimizations through barrier and data transfer optimization, a straw-man reference implementation uses short protocol messages for the same purpose.

According to costs and benefits of optimized separate synchronization and copy-free direct deposit data transfers, the Fx compiler achieves its best performance with the DMSG message passing system based on decoupled synchronization and data transfers, rather than with a postal message passing library such as PVM or MPI. Using a global picture of the communication pattern, the communication compiler replaces the synchronization of numerous data transfers by a single more efficient collective communication primitive (e.g., barrier synchronization). The resulting decoupled communication model permits a simplified implementation of the message passing system that is more focused on the data transfers than a general purpose messaging implementation. No buffering services are provided in this simplified message passing library, since the buffer management is taken care of by the compiler and most of the synchronization is done globally by barriers. Decoupled communication clearly puts an additional burden on the compiler or the application writer. While we do not attempt to argue that deposit message passing will simplify parallel programming for a human programmer, we show in the next section that the HPF-style programming model of Fx suits this communication model well and that most of the necessary synchronization information can be inferred from the structure of the application and its high level array assignments.

5.5.3 The myth of overlapping communication and computation

Many papers on latency tolerance cite the need for overlapping communication and computation in parallel programs. A closer look leads to a corollary of Amdahl's Law that limits the benefits of overlapping communication and computation. A recent journal article [87] points out this fact in the following theorem:

Theorem 3 *The maximal benefit of overlapping computation and communication in a code is a factor of two.*

This maximal benefit of a factor of two is only reached in codes that spend exactly 50% in communication and 50% in computation work and that have no immediate data dependence between data elements used in subsequent communication and computation steps. Furthermore such a benefit is only possible if computation and communication use all separate hardware resources within the node of a parallel system. In Chapter 6 we show that no matter how well designed the network interfaces are, the communication will be memory system limited for most communication operations occurring in compiled data parallel codes, since they have strided or indexed access patterns. Unfortunately, a similar memory system limitation is found in the computation work of all but the most regular codes. This renders the assumption of independent hardware resource and distinct functional units for computation and communication as unrealistic.

Detecting data dependences of the computation step is a further problem. The analysis is required to identify the parts of the data that are ready for communication while other parts are still computed on. For a full convolution such as used in FFTs, all output elements of a computation might depend on all input elements and a partitioning of the computation for overlap is difficult to detect and hard to exploit.

A careful review seems to leave the highly praised benefit of overlapped communication and computation as an unrealistic option to hide communication overheads and speed up applications. Checking the the utilization of the memory system, the internal buses and other resources within a processing node, we find bottlenecks for both computation and communication and with them many reasons why overlap will not result in speedups. Checking the compiler technology available we find difficulties in detecting overlap and scheduling overlapped communication.

5.6 Equivalent mechanisms under deposit synchronization model

In Section 5.2 we outlined a mechanism for deposit and fetch message passing. Given the basic synchronization model and the property of well-synchronized programs, we can identify at least two common message passing techniques that are equivalent to deposit or fetch messaging if decoupled synchronization is used. Such an equivalence is under the synchronization model only and does not mean that those alternative mechanisms offer the possibility of direct deposit, an efficient use of the memory system or the same potential performance. A particular coding style for standard postal message passing systems permits an optimization similar to deposit, but the synchronization in advance remains a performance optimization rather than a requirement for correctness. Active messages is another mechanism that works well with decoupled synchronization and data transfer. In fact decoupling or buffering is a requirement for programming active messages properly, otherwise sequential consistency may be lost, or even worse, the active message dumps core on the remote node as it faults trying to jump to handlers that are not yet installed or that are not yet ready to execute.

5.6.1 Systematically posting (split) receives in advance

In many postal message passing libraries send and receive functions can be split into a start_receive, end_receive, start_send, end_send. MPI supported split calls since its initial definition; PVM added split calls only recently to some implementations.

The split calls permit a user program to supply the bulk of the storage space for buffering instead of relying on the internal buffers typically provided through the message passing system. Also split calls make it possible to designate multiple messages as "ready to go" (split sends) or as "ready to come in" (split receives), thereby improving messaging efficiency on systems that can send and receive messages in/from different directions.

A split send and receive API (application programming interface) is just a primitive means to delegate the responsibility of managing buffers to the user programs. As interface between the library and the programmer they permit a programming style that avoids copying data into system buffers in many but not all cases. If we examine the semantics of both parts of a split call and determine the precise restrictions on the use of the buffers between the two split parts of the call, it becomes apparent that there is no fundamental change in the requirement for buffering but that a well written application program can manage its own internal data structures to provide the message passing system with the necessary buffer space. The complicated guidelines for the proper use of split receive reveal that the fundamental problem of tied synchronization and data transfer is not solved by the introduction of split calls alone.

The use of split receive to designate multiple messages as outstanding can enable performance optimization based on a more direct critical path for message handling. While blocking calls allow for only a single outstanding send and receive, split calls allow multiple receives to be pending at the same time. With the insertion of a decoupled synchronization step (based on a barrier or a separate message class) the case of a pending receive upon message arrival can be made the standard case rather than the exception. Once all receives for data messages are systematically posted in advance the system buffer space can be set at a minimum.

Systematically posting split receives in advance is a programming style that can be used with any standard message passing library such as PVM or MPI, and can achieve some benefits of deposit messaging without changing to the programming interface. The synchronization model of this programming style is identical to deposit messaging since it requires decoupled synchronization and data transfers and the transformation of the application code from a flow-synchronized into a well-synchronized program. We evaluate the benefits of this programming style in a grand challenge application initially coded for standard MPI and then coded for MPI with split receives, systematically posted in advance. The code is executed on a Cray T3D and communication performance improves up to a factor of three (see Section 9.1.7).

5.6.2 Active messages

The mechanism of active messages [108] emphasizes fast processing at the receiver node but does not define all the constraints that programs have to obey to execute correctly. Despite some caveat warnings (e.g. you can not send messages from a handler) neither the original active messages proposal nor most of the follow on discussions [109] suggest a message passing model for active messages. Instead the work focuses and improves mechanisms for message handling without discussing the synchronization model or suggesting a buffer management strategy; these issues are left to the user or a compiler (who has to guarantee, e.g., that the handler invoked in response to a message does not overwrite data that are still "live").

The concept of well-synchronized message passing programs does not endorse a particular mechanism for data transfers such as implicit or explicit receives and leaves the mechanics of the data transfer open, as long the data transfer is pre-synchronized and the data has a place to go when it arrives. Therefore active messages can be seen as an elegant way to post receive operations ahead. The installation and the activation of a message handler (not to be confused with invocation) must be treated like posting a set of receive calls collectively in advance. The receives posted are those for a whole class of messages sharing the same handler.

Theorem 4 *Well-synchronized programs (and only well-synchronized programs!) can be correctly executed using active messages.*

The first part of this theorem is easy to verify. If a message program is well-synchronized, it can be executed through the mechanism of active messages. A well-synchronized programs knows the final data location for every message before the send operation is executed. A simple handler can therefore store the data in its final location.

The second part of the argument, that only well-synchronized programs can be executed by a basic implementation active messages, is quite easy to follow. The introduction of active messages in the original paper [108] states as a caveat that a handler can not depend on further sends or anything that generates network activity. A handler must be able to process the data unconditionally and store the result without any precondition. Thus this requirement is equivalent to designating storage or posting a receive operation ahead. This requirement holds also for non-basic handlers that perform computation, since they are not allowed to block and wait for other operands or wait for a memory location to become available for storing results.

The truth of such an argument is obviously subject to a critical review of what mechanisms are considered part of a basic message passing system and what mechanisms remain part of the application program. The observation applies only to systems that use active messages directly as the primary messaging mechanism. In several proposed active-message-layers [75, 106] the active messages mechanism has been combined with a set of standard handlers that allocate and manage buffers. Assuming infinite buffers, such a system implements a postal model fully compliant with standard postal messaging semantics or even standard connection semantics such as TCP/IP. However, the main benefits of active message are lost in those cases and the structure of those implementations are similar to what vendors like Parasoft (Express) [21] or Intel (NX) [85] provide since the earliest releases of their messaging libraries. Once the buffering is introduced to form a complete active message layer the simplicity and the main advantage of the active messages seems to be gone [63].

5.7 Compiler optimization involving synchronization

A number of previous studies investigate whether more or less synchronization results in faster parallel programs [37, 61]. Previous work resulted in a few optimization techniques that either add or remove synchronization to a parallel program to gain speedup in a particular situation. Adding and removing synchronizations has been proven as a viable optimization in some cases and a generalized answer to the question of whether more or less is better can not be given. Even within the Fx compilers communication back-end examples, both types of optimizations are applied. We give two examples here: There is a communication optimization that improves performance by reducing the amount of synchronization (eliminating redundant barriers), and there are other optimizations that improve performance by increasing the amount of synchronization (inserting additional barriers into a dense communication pattern for improved congestion control in the network).

Elimination of barriers for fully connected communication patterns

Some interesting communication optimizations can be performed in communication steps with fully connected communication patterns.

Definition 1 *A communication step is said to be fully connected (or total) if and only if the transitive closure of the dynamic communication graph from each processor reaches all other processors.*

Such is the case in all reductions, and also in symmetric communication patterns involving all nodes. There are two optimizations that can use the presence of a fully connected communication pattern. The first optimization is the elimination of redundant barriers, and the second optimization is barrier-less combination of data transfers with so called double buffering. A detailed description of these techniques contains the necessary data dependence analysis and measures their implementation [53].

At a first look, both optimizations seem to reverse the separation of control and data transfers. They piggy back implicit control information (e.g., on the availability of data elements or empty buffer slots for future steps) onto the data messages of a current communication step. Due to the global reach of a fully connected pattern, a data transfer can contain a sequencing number and signal control information to the other nodes. The implicit control information can be used as clearance for a future total communication step. Synchronization-free deposit transfers are also used in the double buffering algorithm when two subsequent total communication steps are found to operate on completely distinct sets of distributed arrays or compiler temporaries.

The success of such an optimization does not necessarily invalidate our claim regarding the benefit of decoupling. In both cases the re-joining of control and data transfer and elimination is done only after decoupling them first and after a global analysis. A data transfer per se is not burdened with its own control transfer operation but with a synchronization operation of the the next communication step and therefore the desirable coarse grain pipelining of control and data transfer takes place.

Congestion controlled communication

On parallel supercomputers with adequate hardware support synchronization can be quite cheap. Therefore it is advantageous to insert additional synchronization steps to achieve a better utilization of network resources, such as intermediate buffers for routing. This optimization breaks with the intent of this thesis to treat the network as a black box and distracts from the immediate focus on hardware support in the network interface and message passing software issues at the sender or receiver. Still, if the network is sufficiently documented the benefits of congestion controlled routing on dense communication patterns can be quite significant [55]. In practice this optimization is not without difficulties. The internals of the network routers must be know to the compiler and the optimizations fail once fault tolerance or service nodes introduce irregularities into an otherwise regular topology [99].

The optimization of congestion controlled routing through additional synchronization steps is worth mentioning here because it constitutes a case of improved management of buffers in user space. Taking over complete control of the router puts not only end-point buffers but also switches and network buffers under compiler control. The success of this technique for performance optimization only emphasizes the need to decouple control and data transfers and to manage them separately to the benefit of communication performance.

Chapter 6

Copy transfer model of memory performance

In Chapter 5 I argue that control and data transfers must be decoupled and optimized separately to achieve optimal communication performance on private memory architectures, and I explain how control transfers are optimized in terms of algorithms and implementations. This chapter is dedicated entirely to the data transfers. The *copy transfer model* serves as a tool for analyzing and optimizing the performance of data transfers given a memory system of a particular parallel system. The model is well suited to capture the "per byte" costs in communication work of parallel programs and to compare the communication performance of different hardware and software architectures. In the context of a parallelizing compiler that generates communication code directly, rather than using standard libraries for communication, a good cost benefit model is required to guide communication code generation towards maximal efficiency in memory operations and towards the best possible ordering of communication schedules and loop nests within the communication steps.

6.1 Memory reference patterns in compiler generated code

To map an application onto a parallel system, the compiler must determine how data and computations are to be distributed across the nodes of the parallel system. Recently, the High Performance Fortran (HPF) effort has resulted in a set of *user directives* that assist the compiler in performing its tasks[41][1]. HPF focuses on block-cyclic distribution of arrays, where the two variants, the block distribution and cyclic distribution are the most common [96]. The distributions included in standard HPF language implementation are well-suited to describe regular data layouts. Figure 6.1 shows a copy from cyclic to block distribution, involving only regular accesses. However, many applications are *irregular* in the sense the access pattern cannot be described with a few parameters. Instead, the access pattern is contained in another data structure, usually referred to as an index array. A typical example is `A[1:n] = B[X[1:n]]` where `X` contains some permutation of `1..n` (i.e., there are no duplicate entries in `X`). A great deal of compiler effort is required to deal with the complexities of such code; after all, `A`, `B`, and `X` might all be distributed over multiple nodes. However, the bottom line is that the compiler at some time has to access the elements of `B`, using some intermediate index array `T`, as depicted in Figure 6.2.

From a compiler's point of view, data is moved between the user address spaces of nodes, and these data can be contiguous blocks, slices, intersections of slices[96], or irregular blocks of data described by

[1]My work is done in collaboration with the implementation of an HPF compiler[48], but the details of HPF are irrelevant to this study. My results apply to any system that moves data from the local memory of one node to the remote memory of another.

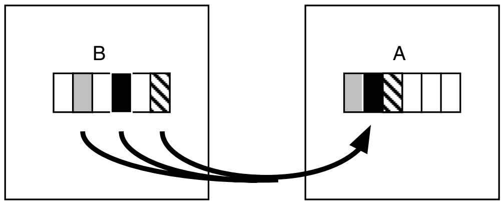


Figure 6.1: Access to block-cyclic, regular array.

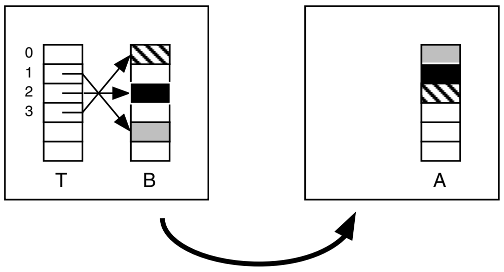


Figure 6.2: Access with an index array.

an index array. The compiler generates synchronization (or control) instructions separately (e.g., before and after a complete array redistribution) [100]. This organization allows us to focus on speeding up the *data transfers*. There are two principal approaches to organizing the data transfers. Either the compiler invokes communication operations for large data blocks as provided by a conventional message passing library - or the compiler uses a collection of remote stores to “put” the data to their destination. Compilers can usually generate the addresses for the loads and stores accesses of such a transfer on either node, the sender node or the receiver node.

6.1.1 Three common addressing patterns

The node programs generated by a compiler to transfer data attempts to take a number of factors into account: the specific data distribution, the size of the array (if known), and the size of the parallel system (if known). For this study basic unit of transfer is a 64bit word, often labeled as a double precision floating point number. If elements of a different size e.g. 32bit or 128bit, are more relevant to an application, it is quite easy to change that parameter. From the perspective of the memory system of a node, three different types of memory access are used in support of communication:

Contiguous All memory accesses are to a contiguous block of data. A contiguous block access pattern commonly results from *block distributions*.

Strided The memory accesses are done to strided data words or strided blocks of data words (e.g., 2 words for complex numbers, 6 words for 3D tensors), with a constant stride $s \geq 2$. This pattern results, e.g., from *cyclic* or *block-cyclic* distributions.

Indexed An arbitrary sequence of non-contiguous words is accessed. The specific array access pattern is determined by indices given in a separate index array. Reading the indices is seen as overhead, so reading the index is considered to be part of the memory access operation and does *not* contribute towards the effective memory access bandwidth reported and the byte count of auxiliary loads for indices are not included in the measured bandwidth figures. Indexed patterns are common for irregular distributions and sparse matrix representations[89].

Strided accesses do primarily occur in assignments between cyclic or block-cyclic distribution, but is also possible that they result from a redistributions between blocked distribution.

6.1.2 Memory systems of parallel computers

In this section I present a simplified view of the node architecture that focuses attention on the basic architectural components relevant to my parallel compilation model. For a first, most basic model I assume a local memory system with a primary cache in the microprocessor and a DRAM-based memory system. I also assume that data are sent and received through a simple transfer to the network interface (e.g. load/store to a FIFO). For the model it is important to capture sequential and parallel operation precisely. Parallel operation occurs when additional functional units are capable of doing memory operations without the involvement of the main processor. This could be DMA controllers, fetch- or deposit engines that process incoming get and put requests without the involvement of the CPU of a node. In Section 8.2 of Chapter 8 I give the details about the memory system hardware of the three parallel machines under consideration, the iWarp, the Cray T3D, and Intel Paragon. In the Chapters 7 I describe the best implementation of direct deposit and finally in Chapter 9 I can relate the deposit mechanisms to the actual hardware instrumentation used to execute communication related memory accesses and to the performance achieved.

6.2 The *copy-transfer model* for communication system performance

Even a simplified view of the memory system offers a rich set of choices to a compiler to organize the inter-node communication. The objective of a compiler is to obtain some highest possible communication performance for a given transfer with the node to node communication pattern and memory access pattern required by a parallel program. In this section I introduce a simple analytic model to reason about different sequences of operations involved in such data transfers. This model can be used to estimate the maximal transfer performance (throughput) as well as to determine rules for generating the best code in the backend of a parallelizing compiler or guide human programmers towards the best implementation.

6.2.1 A throughput-oriented model

Nodes of massively parallel computers typically have just one level of cache and a fairly low cost memory system. This organization is mandated by the pressure to keep the cost of the nodes down. The cost of n-way interleaved or highly banked memory systems, as they are common in vector supercomputers, seems to be too high for a node of a massively parallel machine.

In general the performance of cached memory systems cannot be specified by memory access bandwidth and latency alone. The memory system performance critically depends on temporal locality i.e. on data reuse in the caches. Traditionally the need to accurately analyze the memory system performance for compilers leads to complicated trace driven simulations of the memory hierarchies with caches. With my model I make an attempt to simplify, but still derive useful information from a few characteristic performance figures. In summary, operand reuse and temporal locality work well to improve the performance of computation if blocked algorithms or optimized kernels (like BLAS3) are used. In contrast to the computational work I observe that temporal locality plays only a minor role in the memory accesses for communication. A *throughput* oriented model seems to fit the need much better. A purely throughput oriented cost/benefit model is easier to use for a compiler writer than memory access traces. It might miss some effects but nevertheless reflect the performance experienced by applications well enough to guide performance optimizations.

The importance of throughput for long stream of memory accesses is not surprising given the properties of communication related loads on the memory system. In compiled data parallel programs, parallelism is exploited by operation on large collections, with the data distributed over a large number of processors[11]. In practice, these collections can be quite large and a compiler cannot assume that the local data structure on any node fits entirely into the local cache of a node.

The large amount of data involved in realistic applications further implies that many elements need to be exchanged between any two processors in a communication step. Once the elements for a remote operation are determined, and the communication is started, the transfer mainly depends on the maximal throughput of that copy transfer as a whole rather than on the latency and overhead for transferring a single element.

While the temporal locality does not influence the performance of communication related memory transfers, the spatial locality is an important factor. Some memory systems perform contiguous accesses faster than strided accesses, and strided accesses with constant strides are again performed faster than accesses with arbitrary strides supplied from an index array.

6.2.2 Basic transfers

A compiler-generated communication operation can be decomposed into a few basic transfers or basic steps. I now introduce some terminology to capture the key aspects of these basic steps, which concentrate on common access patterns encountered in parallelizing compilers. A transfer T moves data using a source pattern r and a destination pattern w. The source and destination patterns capture the memory access patterns, i.e. how the data are read and written. The read (load) and write (store) locations are always on the *same* node, unless explicitly noted. To concisely represent such a step T, I mark the read pattern as a *left* subscript and the write pattern as a *right* subscript, i.e. ${}_rT_w$. Typical patterns are 1 for contiguous accesses, $2, 3, \ldots$ for strided access with constant strides of $2, 3, \ldots$, and ω for indexed accesses. I use the access pattern 0 if the source or destination is a register or a fixed location in memory. Its use depends on whether I characterize a communication related or a local transfer. In the case of communication related memory accesses, loads and stores to a fixed memory location (e.g., the head or tail of a FIFO) make sense, since every access the same location moves a new data element from/to the communication system. In the case of local transfers the directing the load or store stream to/from a register file permits to measure a isolated stream of memory accesses as "half" of a copy transfer. For a full copy transfer a load and a store stream must be combined, but the notion of a "half" transfer captures the essence of memory accesses in computations, where a stream of operands is actually consumed by arithmetic units. Load-only and store-only transfers are a also useful to separate the performance aspects of load and store performance during a particular copy.

The basic transfers necessary to perform the communication operations demanded by a compiler comprise local, intra-node transfers (from memory to network interface, from the network interface to memory) and remote, inter-node transfer (across network links):

${}_xC_y$ **local memory-to-memory copy** This transfer is characterized by a read access pattern, x, and a write access pattern y and includes all possible access patterns for reads as well as for write, so x and y can assume values of 1 for contiguous, n for strided, or ω for indexed accesses. The transfer is realized by an optimized (i.e. unrolled and optimally scheduled) load/store loop, executed by the processor to allow general access patterns.

${}_xS_0$ **load-send** This basic transfer copies data form the memory system to a fixed communication system port. The communication port is a constant location, e.g. a FIFO. Since the accesses are done by the processor, x can be any access pattern.

${}_xF_0$ **fetch-send** This basic transfer is similar to the basic load-send operation, but the fetch-send is performed *in parallel* in the background by additional hardware, such as a DMA or fetch engine. There may be restrictions on what read access patterns x are allowed by an implementation, but at least contiguous or constant strides are usually included.

${}_0R_y$ **receive-store** This basic transfer corresponds to the load-store transfer. This transfer accomplishes a copy of data from the communication system into the memory and is performed by the processor. Therefore, y includes the full range of possible access patterns.

${}_0D_y$ **receive-deposit** This basic transfer on the receiver side corresponds to fetch-send. On some architectures, incoming messages can be automatically received in the background, without involvement of the processor. Some systems can handle any access pattern by processing address-data pairs received from the network, while a simple DMA engine puts a restriction on the access pattern y.

These are the basic *intra-node* transfers. To accomplish inter-node communication, data must traverse the network. I distinguish between two network transfers since various parallel systems deal with these two cases differently.

N_d **data-only in network** The N_d transfer moves data words only - no addresses across the network.

N_{adp} **address-plus-data in network** The N_{adp} transfer captures those inter-node transfers where a remote store address is sent along with the data. Depending on implementation details, these remote store addresses can be passed along as "address data pairs" or compressed as addresses for a block of data. However, most current keep the hardware as simple as possible and choose the address-data-pair variant, if they support this transfer at all.

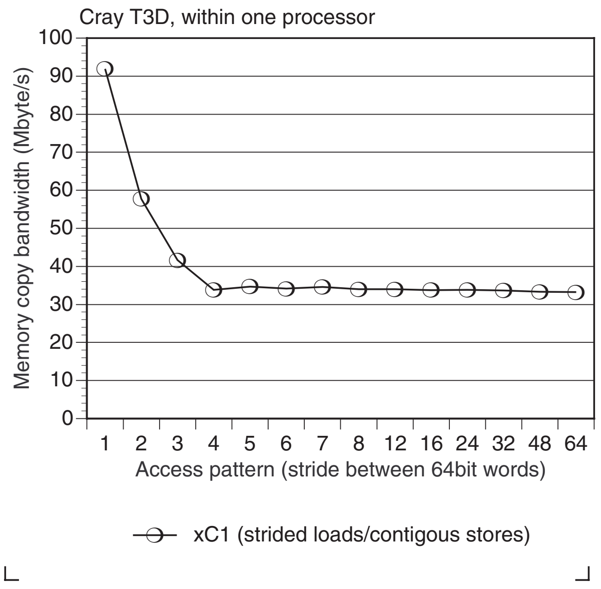
The informal rules of my model decides the type of network transfer based on the subsequent receive. It is further assumed that the sender determines the addresses or at least the mode of addressing. A transfer can be contiguous, so transferring a single base address and the whole data block is sufficient; it can be strided and therefore depend on the hardware whether it can handle that form of strides or it can be indexed and an address word has to be sent with each data word. This distinction of network transfers is made because in most networks the effective bandwidth for data alone is different for transfers with address and data.

6.2.3 Example: T3D in the basic copy transfer model

For the local, memory-to-memory copies all combinations of different load and store access patterns make sense, and their performance can be measured by micro-benchmarks. Based on observations I find that the entire space of source and destination patterns is sufficiently characterized by two performance graphs, the graph of a gather copy ${}_xC_1$ (strided loads, contiguous stores) and the graph of a scatter copy ${}_1C_y$ (contiguous loads, strided stores) as shown Figure 6.3. Cases of memory copies with non-unit load strides and non-unit store strides are encountered very rarely, and graphing the full space in multi dimensional charts seems unnecessarily difficult for the presentation and for the discussion of a memory system. Still the model captures all data points of the full parameter space and an extended table with benchmarked performance data can be made available to the compiler in case that some compilation technique finds some interesting tradeoffs based on one of those additional data points.

The performance spectrum of the different access patterns in transfers to and from the network interface contains much less variety than the performance spectrum of local accesses. The reason is that on most parallel machines I/O related accesses are neither cached nor pipelined. Thus network related performance characteristics can be captured in simple tables rather than in sets of graphical figures. My example in Table 6.1 shows the transfer performance of the network related accesses in an Intel Paragon node. Note a distinction between DMA accesses (performed in the background) and processor driven accesses (performed in the foreground). The distinction of transfers according to the agents available (processor, DMA, deposit engine) helps to determine whether the basic transfer steps are composable in parallel or by sequence.

For the characterization of the basic transfer across the network wire, the model distinguishes two modes of data transfers and a few characteristic constant factors of congestion. The modes of data transfers include data elements only and address data pairs (addresses sent along with the data words). The



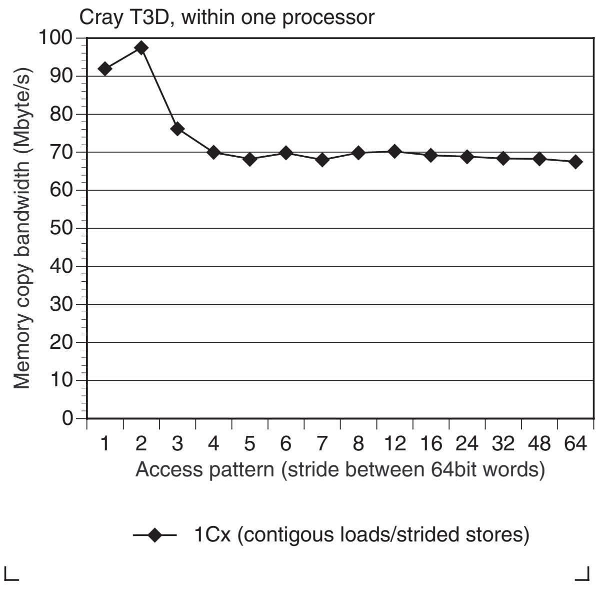


Figure 6.3: Measured performance of the local memory system for large transfer, with either strided loads or strided stores for the T3D.

a	b	$\vert {}_1S_0\vert$	$\vert {}_1F_0\vert$	$\vert {}_{64}S_0\vert$	$\vert {}_\omega S_0\vert$		
Paragon	Send	52	160	42	36		
		$\vert {}_0R_1\vert$	$\vert {}_0D_1\vert$	$\vert {}_0R_{64}\vert$	$\vert {}_0D_{64}\vert$	$\vert {}_0R_\omega\vert$	$\vert {}_0D_\omega\vert$
Paragon	Receive	82	160	38	-	42	-

Table 6.1: Measured performance of transfers between the local memory system and the network interface for the Intel Paragon with processor driven and DMA transfers. Send related transfers (left) and receive related transfers (right)

congestion factors indicate the average per link congestion of a particular communication pattern. The assumption of a fixed congestion is justified in Section 9.2.3 along with a discussion of the network topologies used in the three parallel machines under investigation. Table 6.2 shows the network bandwidth of a Cray T3D as an example. The most common congestion factor (2) is emphasized in boldface.

	Average congestion					
	data only (N_d)			address data pairs (N_{adp})		
	1	**2**	4	1	**2**	4
T3D	142	**69**	35	62	**38**	20

Table 6.2: Network bandwidth (MB/s) as a function of a fixed overall congestion.

As previously stated, the performance figures in the basic copy transfer model specify the asymptotic peak value for large transfers. It is tacitly assumed that the number of elements transferred exceeds the sizes of all caches, write-back queues and other buffers along the data path.

6.2.4 Estimating throughput for communication operations

After defining the basic transfers, a user of the model can start to compose communication operations for a variety of access patterns simply by concatenating basic transfers. There are two concatenation rules and operators: Two transfers, that use the same resources (e.g., the processor) must be concatenated in sequence $\circ$. The write (left subscript) access pattern of the first transfer must match the read (right subscript) access patterns of the second transfer, e.g., ${}_0R_1 \circ {}_1C_{64}$. Transfers that use disjoint communication resources can occur in parallel $\|$.

The decomposition of the communication operations as basic transfers permit to estimate the maximal throughput of a transfer for several different implementations of a particular communication operation. I use the following three rules to derive an estimate for the effective throughput $|Z|$ of a communication operation Z based on the throughput of the basic transfers involved.

$\|$ **Parallel composition** If two transfers occur in parallel, the composite throughput is the minimum of the two throughput figures, i.e. $|Z| = min(|X|, |Y|)$.

$\circ$ **Sequential composition** If two transfers cannot occur in parallel because they share a common resource, the composite throughput is the reciprocal sum of the two throughput figures, i.e. $|Z| = 1/(1/|X| + 1/|Y|)$.

$<$ **Resource constraint** For some more advance performance estimates, the basic model can be extended with some additional resource constraints, that reflect resource limits preventing an increased throughput of transfers that can occur in parallel. For example, if the processor and the DMAs share a common system bus, the total bus bandwidth cannot be exceeded and the two transfers can not add up to more than this limit. Resource constraints are specified as inequality in bandwidth variable. If a resource constraint cannot be met, the throughput parameter of the participating basic transfers must be reduced until the constraint is met.

Example: Buffer-packing transfers, PVM style

The most performance critical communication operation in a communication code of a parallel compiler is a local memory to remote memory copy ${}_xQ_y$. Depending on the distributions of the array operands of an array assignment, different access pattern may be encountered for load accesses (x) at the source and store accesses (y) at the destination. On different architectures the transfers might be carried out by different mechanisms in hardware. To give an example I formulate a buffer packing transfer in the basic copy transfer model and estimate its maximal achievable throughput.

Formulation of the transfer in the model

${}_xQ_y$ captures the most general, data intensive communication operation, performed by a compiled program. One way to implement this operation is to perform a local "gather" copy operation C that reads the items to be transferred and stores these data into a contiguous block of local memory. Then this block of data is transferred to the network interface (i.e., a load-send S is done), followed by a network transfer N. On the remote node, the data are extracted from the network into some buffer (via a receive-store transfer R or via a contiguous deposit-store D), and a final "scatter" copy C moves the data to the intended location. I call this implementation of ${}_xQ_y$ *buffer-packing* communication, here written as a concatenation of basic transfers:

$${}_xQ_y = {}_xC_1 \circ ({}_1S_0 \| N_d \| {}_0D_1) \circ {}_1C_y$$

It might appear that for contiguous transfers (${}_1Q_1$) the first and the last memory copy (${}_1C_1$) are unnecessary. This is true in principle, but message passing libraries like PVM force the programmer/compiler writer to copy the data elements in all cases to comply with the standard application programming interface and as an issue of buffer management. Of course, there may be different ways to implement ${}_xQ_y$, especially if constraints are placed onto x and y. Therefore I return to this topic in Section 9.2.4. The goal of the modeling is to obtain the throughput figure of interest for all communication operation composed by the compiler from the basic transfers.

Estimating the resulting throughput

The simple technique of concatenating transfers works, since the same number of data elements is moved through all steps of a communication operation. As an example I estimate the throughput for a conventional message passing operation with buffer packing on the T3D. The transfer involves an array transpose of an $n \times n$ array (i.e., `b[i][j]=a[i][j]`). The first case captures the behavior of the program fragment using the vendor-supplied custom PVM library, the latter case is an example of the communication operations produced by expert programmers or high-quality compilers using the direct deposit model. The access pattern results in blocks of contiguous loads and strided stores, i.e. ${}_1Q_n$.

I compute the bandwidth by applying the bandwidth rules to my formulas for contiguous transfers. For buffer-packing message passing I obtain:

$$|{}_1Q_n| = \frac{1}{\frac{1}{|{}_1C_1|} + \frac{1}{\min(|{}_1S_0|,|N_d|,|{}_0D_1|)} + \frac{1}{|{}_1C_n|}}$$

For many patterns, e.g. next-neighbor or all-to-all personalized communication (AAPC), every node is sending and receiving at the same time. Therefore I must check that the memory system store bandwidth of the parallel send and receive operations does not exceed the total memory bandwidth ($|{}_0C_x|$).

$$(2 \times |{}_xQ_y|) < |{}_0C_x|$$

Evaluation of this formula with the numbers for a transpose of a 1024 x 1024 matrix on the T3D results in:

$$|{}_1Q_{1024}|_{est} = \frac{1}{\frac{1}{93} + \frac{1}{\min(|126|,|69|,|142|)} + \frac{1}{|67.9|}} = 25.0\text{MB/s}$$

For comparison, measurements of the same communication operation on a 64-node T3D yield

$$|{}_1Q_{1024}|_{mes} = 20.0\text{MB/s}.$$

6.3 Local and remote memory accesses

On machines with hardware support for a global address space the copy transfer model describes local and remote memory accesses in uniform terms, namely as ${}_xC_y$ or ${}_xQ_y$. The latter notation for a remote (inter node) memory transfer is a concise parametric description for various types of transfer option. The model notation of such a transfer is independent of the underlying mechanism, whether it is done by a cache coherence protocol of a shared memory machine with implicit communication, as a simple load/store copy loop on a machine with explicit communication or even as communication subroutine involving buffer packing and calls to message passing library. Although the model encourages the decomposition of the transfers into primitive operations, it provides a uniform notation to describe the entire transfer in all cases considered in this thesis.

Due to this uniform notation the copy transfers model becomes an ideal tool for the characterization, description and graphical illustration of the memory system performance in all classes of parallel systems. The model is capable of describing all transfers as ${}_xA_y$ using the same parameters, namely load/store access patterns x and y, and mode of operation, namely processor or DMA. Despite this uniform notation the detailed formulation gives a convenient terminology to explain performance differences and to describe different machines. Different letters for different kind of transfers (i.e. C and Q) also help to maintain the distinction distinction between local and remote memory accesses although the characterizations look very much alike.

For symmetric multiprocessors systems with separate CPU boards and memory boards the terms local and remote need some clarification. I call all memory accesses that go to caches or to DRAM memory within the same processing node or the same processor board *local accesses*. The remaining memory accesses cause data to traverse at least one processor- or memory board boundary. They may be classified as local or remote accesses. The distinction is made based on the reading processor (consumer) and the writing processor (producer). If the reading and writing processor of a copy transfer are the same, the purpose of the memory operation is primarily intermediate storage, and I classify such an access as local, despite the fact that the data crosses the system bus or a dedicated switching crossbar to reach the corresponding memory module. In contrast, if the reading processor and writing processor of are different for a copy, the purpose of the memory operation is inter-processor communication and therefore I call such transfers *remote accesses*. Such a definition of local and remote accesses in a machine with coherent shared memory follows from the owner computes rule, which is upheld in my compilation model, on MPPs as well as on SMPs.

In the machine characterizations and practical performance evaluations found in Chapters 8 and 9 I often refer to the maximal throughput of local loads or local stores as local memory system performance or local memory copy bandwidth. Similarly the throughput of remote loads and stores is referred to as communication performance and the maximal throughput of copy loops with remote accesses can be called a remote copy transfer bandwidth.

6.4 Extended copy transfer model

One of the basic assumptions of the basic copy transfer model must be revisited for the analysis of shared memory machines and of workstations with multi-level memory hierarchies. I initially stated that the memory operations within communication work involve large amounts of data and can not benefit from data reuse in caches, i.e., the operations have no temporal locality. This observation might not be true in future machine with large caches.

Extending a model with additional parameters is a difficult task and as such it must be done with caution. The increased number of parameters permits more accurate modeling and an improved coverage of the reality, but more parameters mean also more complicated manipulations and intractable calculations with the model equations. A single new parameter can modify a model to such an extent that its elegance is lost, and that the "enhanced" model becomes less useful despite its better coverage. At this point I can not make a no final conclusion on whether working sets make the basic copy transfer model intractable or make it loose its simplicity, but I have two good reasons at hand to justify its revision. The reasons also explain the sudden invalidation of my initial assumption that working sets of the modeled memory operation are are always quasi infinite for copy transfers and not necessary in the model

The assumption of infinite working set certainly holds for machines like the iWarp, T3D and Paragon because those processors contained only a few kilobyte of L1 cache or higher level caches at all. A more recent generation of parallel machines, including the T3E and DEC 8400, have larger L2 and L3 caches - up to a few megabytes in size. With those caches it could become conceivable that properly blocked algorithms would not only speed up computation operation like BLAS3 kernels, but that communication operations with data transfers between processors like transposes could be blocked for caches and accelerated. Assuming a massively parallel system with a dozens of processors such blocked communication becomes realistic only if the ratio of the cache size versus the total numbers of processor becomes large. Only with a large ratio the transfers block between any two processors might be large enough to amortize startup overheads for multiple communication operations to make blocking worthwhile. However the fixed size strides of accesses in transposes do not execute well on memory systems with direct mapped caches an not even with n-way associative caches, unless the data structures are laid out in a perfectly skewed order. So at this time is not clear that cache to cache communication operation can be utilized by a communication code generator of a parallelizing compiler in many relevant cases, but it would certainly be wrong to ignore this opportunity for optimization due to an insufficient underlying machine model.

Another equally compelling reason for the extension of the copy transfer model with working sets, is its potential use for memory system characterization in general. Initially the basic copy transfer model served as a tool to describe and optimize the internal data path of memory based communication operations. If helped to investigate the maximal performance of different alternatives to pack buffers and to move data from user memory to system buffers and finally to the network. Looking at a larger number of memory systems I discovered that simple copy transfers with their access patterns as parameters are extremely useful for the discussion, the comparison and the design of memory systems per se, not just for the properties relevant to communication work. With a wider application of the model in mind, the cache structure of a memory hierarchy becomes an integral part and must be incorporated in a performance characterization.

These two reasons lead to the introduction of a new working set parameter into the copy transfer model, in addition to the existing parameter for the access pattern and the mode of operation (processor or DMA). The resulting model is called the extended copy transfer model.

6.4.1 Working sets

The extended copy transfer model captures the effects of temporal and spatial locality separately in two parameters. The new working set parameter describes the degree of support for temporal locality and the standard access pattern parameter (stride) captures read-ahead and other support for spatial locality. In the extended model it is no longer assumed that all transfers are arbitrarily large. The working set means that a transfer affects only data within a given working set. If the underlying memory system architecture has the capability to cache that working set at a higher level of the memory hierarchy, the better access performance of that level of memory (L1, L2 or L3) is reflected by the characterization rather than the lowest level of DRAM performance. The micro-benchmarks are take with fully initialized and primed working sets, i.e., warm caches and therefore the extended model does not incorporate any fixed startup overheads or per element latencies.

In benchmarks the whole working set of data is transferred even when strides are involved. The strides just determine the order in which the elements of the working set are accessed and transferred e.g. working set of 20 with stride 5 would order the accesses as follows: $\{0,5,10,15,1,6,11,16,2,7,12...19\}$. The model remains a pure bandwidth oriented model since all transfers are still performed under the assumption that the transfer itself remains large relative to any constant per message overheads. If such constant overheads interfere with measurements in practice (e.g., for measuring on conventional message passing architectures with high setup costs) the benchmarks are forced to transfer the working set over and over again until constant overheads become negligible and true asymptotic bandwidth numbers are obtained.

In theory the effects of spatial locality and temporal locality are orthogonal. Any working set can be combined with any stride, at the source and the destination end of each transfer — independently. In practice some unwanted interactions between the stride and working sets are encountered. In general working sets must be larger than the strides or otherwise loop overheads may become visible and make a direct measurement of performance difficult. There are also interactions between the accesses of the load stream and the store stream in every local copy. In those copy transfers the stores might gradually evict the loads from the working set or obscure the performance picture otherwise. This problem does not exist for remote copies in most explicit load/store architectures since remote and I/O loads and store do not share any caches with the local memory operation in the same basic copy transfer. For local copy performance and remote copy performance in coherent shared memory architectures the interactions are a problem, especially when caches are write back and direct mapped. A proper performance characterization is made with two "half" transfers that originate or end in the register file. A detailed description of the experimental setup for my characterizations of the DEC 8400-SMP (multiprocessor) is found in Section 8.1.

6.4.2 Example: The DEC8400 in the extended copy transfer model

After the introduction of working sets into the model and the modification of my micro-benchmarks the different performance of the various levels of memory hierarchy becomes visible. I pick a DEC 8400 workstation with just one processor for an illustration, because of its interesting four level memory hierarchy. Chapter 8 is devoted to the characterization of five different machines. Figure 6.4 graphs the bandwidth of a simple load stream to the register file — denoted as ${}_xC_0$ in the copy transfer model. A load stream is just "half" a copy transfer since only the load stream affects the memory system, while the store stream is directed towards the register file and used by the floating point unit. With the introduction of working sets the plain line graphs of in the basic model (bandwidth vs. stride/access pattern) become three dimensional performance surfaces that resemble mountain ranges with slopes and different plateaus for the characteristic working sets of the memory hierarchy.

The performance characterization the extended copy transfer model cleanly separates the performance impact of read-ahead related cache hits from and the performance impact of reuse related cache hits. Read-ahead properties characterize increased performance for spatial locality (dimension contiguous to strided, along the stride axis) and reuse-properties show increased bandwidth for temporal locality (dimension small to large along the working set axis).

The slopes between the different plateaus indicated whether a clear eviction of the working set is taking place (direct mapped caches) or whether I encounter a mixed performance picture with a steady transition (unified caches or caches with high associativity). As an interesting detail in the slopes of performance I observe isolated "ridges" and "ravines" for certain particular uneven strides, where performance falls more abruptly or more gradually to a lower plateau. Typically those ravines represent strides, that are powers of two or otherwise multiples of the number of memory banks, the ridges indicate strides that are prime relative to the number of memory banks. The higher or lower performance for those strides is indicative of cache load performance with or without bank conflicts in memory systems that have multiple memory banks.

6.5 Other approaches to memory system modeling

The traditional approaches to memory system modeling are: various methods based on memory access traces for a most accurate modeling of the memory system access behavior of one particular application and

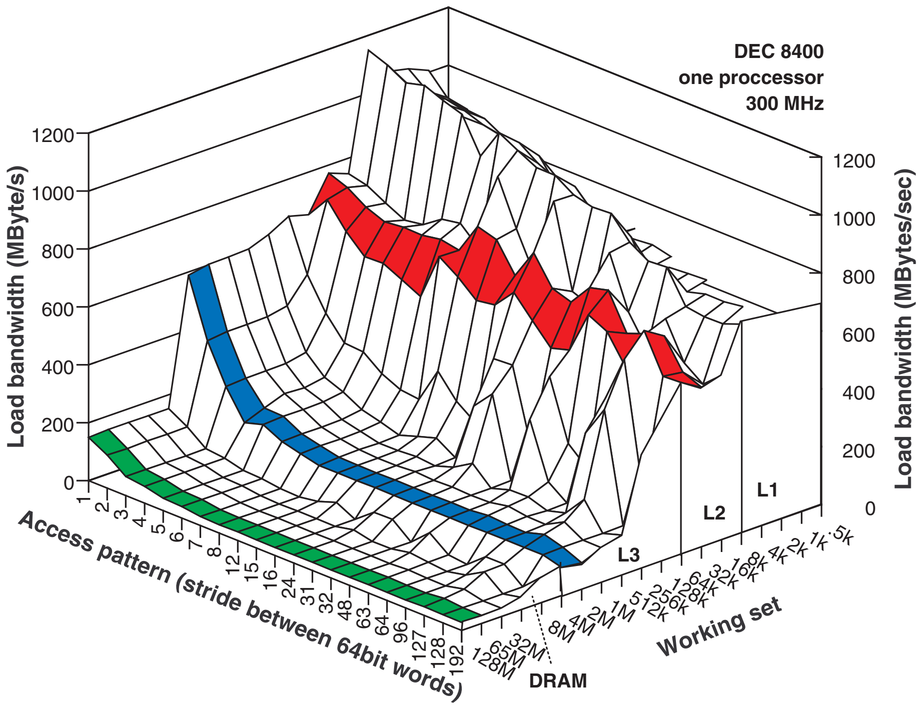


Figure 6.4: Bandwidth of loads for different access patterns (strides) and different working sets on a one processor DEC 8400.

a few high level theoretical models established by the designers of parallel algorithms and by theoreticians in search of complexity classes. Compared to these traditional methods the simplicity of the copy transfer model might raise a suspicion that copy transfers are just a glorified piping analysis combined with some architectural common sense reasoning. I will not attempt to deny this suspicion, but give a brief survey of the alternative models and explain the particular shortcomings that rendered them unusable for my purposes. The copy transfer model wins by its simplicity and its immediate reflection of architectural support provided by the hardware to be modeled.

6.5.1 Trace based modeling

The classic methods for the quantitative analysis of memory system performance in microprocessors are different variations of trace based simulations. An test suite of application is run under instrumentation and all memory accesses are logged for later simulation. The difficulties of dealing with giant traces is overcome in modern tools by a simulation and evaluation module that is integrated into the run-time system of the machine being profiled and integrated [18, 93, 105]. While those techniques are extremely valuable for uni-processors they are inherently problematic with multiprocessors, especially if large amounts of data are being transferred between processors. The problem is that trace driven techniques are intrusive. On uniprocessors intrusion can often be factored out by halting the wall clock while a traced operation is recorded. Profiling is therefore limited to clock cycles excluding those used for trace processing. On a multiprocessor there is coupling between multiple processors. A slow-down of a processor due to profiling might introduce some communication skew and delays that can not be distinguished from regular application skews and delays. A successful profiling effort for parallel programs is reported in [46] even for tightly coupled systolic programs. However a closer examination of the tools in that investigation showed that the programs used in that study are extremely static, and that except for a few trip counts in the loop nests, all the profiling evaluation was done at compile time. The study did investigate tightly coupled compiler generated parallel programs, but on a parallel machine with a static instruction scheduler and a memory system of a single fixed access time and without caches.

Furthermore trace-based approaches require a well chosen set of benchmarks that the are representative workloads, but that are simple enough for the architects to understand. Tracing the memory behavior of real application programs is not of much help if the complexity is so high that the memory system behavior of their parts is no longer understood in detail. Even for uniprocessors the choice of simple, representative benchmarks remains controversial and few people believe that SPEC95 programs should be the yardstick for all architectural tradeoffs and decisions. For parallel programs good standard benchmarks are even harder to come by, although the SPLASH suite has provided a reliable basis of comparison among different coherent shared memory architectures. If run properly, profiled benchmarks give an accurate memory performance comparison. They can give answers to questions on "if" and "why" a particular application is slower on one machine than on another but it remains a difficult task to relate the simulation results directly to architectural deficiencies and possible improvements.

A simple model like the basic copy transfers seem to be more more suitable for reasoning about the high level design choices in a communication architecture, in particular for deciding software architecture tradeoffs (e.g., whether to pack buffers or chain transfers). The simple piping model points directly at the bottleneck or at the particular weaknesses of a memory system and tells where to improve capacities along the data path.

6.5.2 The PRAM model

The PRAM model [40, 88] does neither support temporal nor spacial locality. It would presumably represent the performance of a machine with a single constant figure over the entire parameter space of the extended copy transfer model. Such must at least hold in theory and it would be fun to see how much PRAM like (i.e. flat surface) the characterization of a real PRAM (e.g. the Saarbruecken-PRAM machine) looks like when the micro-benchmarks for an extended copy transfer characterization are run.

6.5.3 Copy transfer vs. NUMA

Several prototypes of early shared memory computers lead to the definition of the non-uniform memory access model (NUMA) and a large number of papers like [113] analyze the performance of several algorithms under that model. The NUMA model does not distinguish between local and remote memory, just between fast and slow memory. The basic problem with the NUMA model is that it takes inverse latency for bandwidth, fails to capture the more subtle cases of pre-fetched data and fails to consider pipelined transfers or performance differences due to strided accesses through caches.

6.5.4 Copy transfer vs. LogP

Some of shortcomings in the NUMA mode are addressed in the LogP model [24]. The gap parameter is nothing less but an inverse bandwidth parameter. Maximal sustainable bandwidth defines the minimal gap between elements, therefore LogP can capture transfer bandwidth independent of the element latency and overhead, but still the model fails to deal properly with access patterns of transfers between a pair of nodes. Coherent and non-coherent shared memory machines show radically different performance for contiguous, strided and indexed accesses even if they incorporate large blocks and the same two nodes as communication partners. Such is due to cache line size and effects of false sharing. The per message overhead of larger messages is considered more appropriately in the LogP model, still it remains quite unclear under which circumstance the element of complex (strided or indexed) transfer is subject to a message overhead. So the LogP model with its fully abstract and uniform unit of transfer, fails to capture some important performance characteristics related to the memory systems and communication operations in parallel machines.

Chapter 7

Implementation

To prove the utility of my concepts I implemented the direct deposit model on several architectures. Some implementations were stable and resulted in released software. In this case the implementations were documented in the release. Other implementation remained unstable, because of too many undocumented hardware *features*. Unfortunately it is quite common that the hardware properties of the network interfaces of massively parallel computers are not completely know or at least not documented to the university collaborators. It is typical that the hardware designer provide many more capabilites than the system software developer are going to use for a production system. Some of this hardware support is found to be buggy and some is found to be ill suited for its purpose. Therefore the hardware software system of production machines uses only parts of the available support and only that part of the hardware support is properly tested. With my implementation I tried to pull in additional hardware and system software capabilities in the hope that they are working correctly. This was mostly the case for the Cray T3D and the DEC 8400. In the latter case the hardware worked flawlessly but the POSIX operating system support for mapping large regions of shared memory posed some problems. In the case of the iWarp component I found a subset of hardware capabilities that worked and carefully documented the capabilites that did not work. On the Intel Paragon myself and collaborator Michael Hemy never managed to program a production quality direct deposit system. The system worked well enough to obtain the performance figures for a solid evaluation but never made it to production quality. Still the implementation design alone resulted in many interesting insights about the distribution of work between a processor and a co-processor.

7.1 iWarp messaging

The deposit message passing library for contiguous blocks of data and its direct deposit extension with address-data-pairs for arbitrary communication patterns were the first deposit messaging systems implemented. Despite the consideration of all principles for a clean deposit model, it it not possible to avoid copies in some cases with the iWarp machines. A description of the implementation follows the necessary communication steps as given in the model section.

7.1.1 Steps of a direct deposit transfer on the iWarp

Note that only data transfer of contiguous blocks can be handled immediately with out buffering in a zero copy style. This possible due to the deposit model with pre-synchronized transfers. Barriers are used to provide synchronization and so there is no buffering of messages that come early.

Sender: Buffer Pack/Local Gather

For the most general case with strided or index access pattern at either end, sender or receiver, transfers must include a buffer packing step. This is due to the processing of send and receive with a single processor on each compute node. The implementation must rely on the spools (DMA agents) to manage the sends or the receives. iWarp spool have limited capabilities and can only transfer contiguous blocks. Therefore strided and index accesses must be handled with a buffer packing step. The buffer packing step at the sender can also compute destination addresses and prepare a block of address-data pairs for direct deposit instead of a data-only block.

Sender: Send Transfer and Send Paketize

On iWarp all programs run in systems mode and there is no paged memory in the compute node. Therefore no transfer step is required. The router of the iWarp component uses wormhole routing, so packets can have an arbitrary length. The headers and trailers can be generated by the low level driver when the data is injected into the network and no specific packetization step is required.

Sender: Send Inject

Data is injected to the network by using the processor for routing headers and starting a spool (DMA agent) for the data body. It is important that the processor never stalls and remains always ready to receive when data is injected to the network. An empty injection queue of eight words is large enough to hold the entire header of a message for all routers used in this thesis including one or two words indicating the total length of the message an a base address for the destination of the data.

As a first operation the output queue is bound to channel leading into the right direction of the 2D-torus. This can only be done when the queue is empty (i.e. all flow control credits came in). The hardware takes care of housekeeping and there is a RESERVE_RPI instruction in the iWarp instruction set to do obtain an empty queue in a particular direction. Once the queue is ready, the message header word, and the peel off routing headers are inserted.

Then a spool (DMA) is started to inject the message body from a buffer into the communication system. Once the spool is started the processor is freed and the communication system works with the memory system to inject the message body into the network. This process can block at any time since the network is reliable and has link level flow control.

Once the entire message body is inserted the completion is detected by polling or alternatively by an interrupt and the processor must inject the message trailer to finish a message transmission. It is important that at no time the processor is completely blocked and that then injection of a body, the injection of trailer, the reception of a body and the reception of trailer can interleave arbitrarily.

Sender: Receive Extract

In the original form of direct deposit the extraction of a message is done in a similar way as it is done in active messages. When a message is arriving from the network an MATCHED_ARRIVAL interrupt is taken an the message is received by the processor in three steps. First the header word is received, then the body is processed using the length and destination address field. The body can be a contiguous data block (deposit) and its data is copied from the network interface to the memory in the efficient data copy loop. There is not problem to process data at the full network speed of 40 MByte/s. If the body is not a contiguous block but rather a block of address data pairs (direct deposit) the handler resolves the addresses and stores just the data to its final location. The handlers can implement a variety of memory operations.

The work on the Catacomb Compiler Backend for the Fx Compiler [95, 94] specifies address-data pair and address-data blocks with different sizes for base data types from a 32bit Integer to 128bit double complex floating point number. The base address could also refer to a table with addresses or indexes instead of specifying the destination. This would be called indirect deposit. For the handlers all memory operations are fair game, as long as they are fast and there is no waiting involved. The handler executes a high priority with interrupts turned off so there are limitation on what can be done in a handler. The direct deposit and direct fetch models suggests that such handlers become implemented in hardware as this will be the case for the Cray T3.

Message handler for a simple deposit of a contiguous block.

```
och_asm:                  -- PART1: message header and body
  bsetcsr 0,mode          -- reenable spooling
  movecsr 0x800,eventc    -- clear matched arr status in event reg.
  bsetcsr ev1,stopcc      -- set the stopcc cond. for queue ev1
  savebind 0x0,ev2        -- save gate0 read bindings in ev2
  bset.w 0x7,ev1          -- set a command bit in ev1 for bind
  bind 0x0,ev1            -- bind  save gate0 read to queue ev1
  movereg ga0,ev5         -- remove the header word
read_body_block:
  movereg ga0,ev3         -- ev3 is the number of bytes
  lshr 2,ev3              -- ev3 is the number of words
  movereg ga0,ev4         -- ev4 is the start address
  loop ev3
    st ga0,(ev4,4)+=
  endloop
  bind 0x0,ev2            -- restore the gate0 read bindings
  retmfe

cch_asm:                  -- PART2: message trailer
  bclrcsr ev1,stopcc      -- clear the stopcc cond. for queue ev1
  savebind 0x0,ev2        -- save gate0 read bindings in ev2
  bset.w 0x7,ev1          -- set a command bit in ev1 for bind
  bind 0x0,ev1            -- bind  save gate0 read to queue ev1
  movereg ga0,ev5         -- remove the trailer word
  bind 0x0,ev2            -- restore the gate0 read bindings
  movecsr 0x400,eventc    -- clear stopcc status in event register
  ldlit32 _dmsg_cnt,ev7   -- record that one more message has arrived
  ld.w (ev7,0),ev6        -- in global semaphore dmsg_cnt
  add.w 1,ev6
  st.w ev6,(ev7,0)
  retmfe
```

Note: Some instructions are pseudo ops that can be expanded by by a macro assembler into 2-3 instructions with scratch registers.

Message handler code fragment for a direct deposit handler (a stream of address-data pairs)

```
read_body_adblkbc2:        -- adblkbc protocol, double-word data
  movereg ga0,ev6          -- ev6: 1=need read a garbage wrd at end
  movereg ga0,ev2          -- ev2 is the number of blocks
  movereg ga0,ev7          -- GARBAGE (filler for alignment)
  movereg ga0,ev3          -- ev3 is the stride
  movereg ga0,ev7          -- GARBAGE (filler for alignment)
  sub 4,ev3
  movereg ga0,ev5          -- get first address into ev5
  movereg ga0,ev7          -- GARBAGE (filler for alignment)
  flags sub 1,ev2
  brrelif ilu.zero,jms_adblkbc2epi
  loop ev2
    movereg ga0,ev4        -- ev4 is the block size
    movereg ga0,ev7        -- GARBAGE (filler for alignment)
    loop ev4
      st ga0,(ev5,4)+=
      st ga0,(ev5,ev3)+=
    endloop
    movereg ga0,ev5        -- ev5 is the address
    movereg ga0,ev7        -- GARBAGE (filler for alignment)
  endloop
```

Receiver: Depaketize, Transfer

The iWarp architecture supports large messages and all operations take place in a single address space for user and system code. There is no need for depacketize and transfer between address spaces.

Receiver: Unpack

If a pure deposit model (contiguous blocks) is used, data must be unpacked by the processor in a copy loop. With advance direct deposit data handlers the addresses go along with the data stream and data can be deposited directly to its final destination - so the unpack operation is included in the receive handler.

Receive - Unpack

For deposit as well as direct deposit with address data pairs the receive operations are highly integrated into a message arrival handler. Whenever a potentially long message arrives at the network interface (one of the incoming queues) and interrupt is flagged and the DMSG message passing system invoked a message receive handler. The processor is completely tied up by receiving a message but sending messages continue to make progress due to the use of spools (DMAs).

7.1.2 Direct Deposit API and Code Example

As a minimal, most efficient direct deposit communication facility it contained the following modules:

```
Module            Function                        Lines of Code
```

Name		C	asm_C_mix	asm
router.c	2D torus router	137		
espl.h	Spool/DMA Control		90	
dmsg_evh.c	Msg Trailer Handler	73	10	
dmsg_evh_asm.s	Msg Arrival Handler			85
dmsg.h	Msg Header Macros		100	
dmsg.c	Init and Sends	365		
Total	Deposit Message Lib.	575	200	85

A simple example shows the API of the DMSG library. The example initializes the library. It uses a global barrier synchronization `dmsg_sync` to resolve control issues like clearing the deposits ahead of the send. The test send one deposit message from node/cell 0 to node/cell 1 synchronizes and then proceeds by a all-to-all personalized communication sending 64 individual messages from each processor to its 64 communication partners (including itself).

```
struct iwcfg cfg;    /* machine configuration and individual nodeid */
float out[1024][64], in[1024][64];      /* output and input buffers */
extern int dmsg_cnt;         /* semaphore for receiver notification */

int main() {
  ...
  size=1000; getcfg(&cfg);
  dmsg_init();
  dmsg_sync();
  if (cfg.cellid == 0) {
    dmsg_send(1,out[1],in[0],sizeof(float)*size);
  } else if (cfg.cellid == 1) {
    dmsg_await(1);
  }
  dmsg_sync();
  for (dst=0; dst<64; i++) {
      dmsg_send_ctrl(dst,out[i],in[cfg.cellid],sizeof(float)*size);
  }
  dmsg_await(64);
  dmsg_sync();
}
```

The released library achieved an overhead of 400 clock (20 μs) and a throughput of 39.48 MByte/s with 65kB messages, which is 98.75% of the maximal bandwidth of the physical interconnect. Roughly half the bandwidth (21 MByte/s) is achieved for messages with 1KByte size.

The library was used in the Fx Compiler whenever an application relied on arbitrary communication pattern with communication all nodes of a larger iWarp system. These were quite a few application, but it must be noted that a fair number of applications for the iWarp systems could be written using only fixed point to point connection or even use fine grain data streams between the nodes according to a systolic algorithm for the given calculation problem. In this case the programmed communication services package [52, 54] was used. With a simplified router the communication resources (16 user queues) could be partitioned between message passing (5 queues), synchronization (3 queues) and connection oriented communication (8 queues). This allowed DMSG to be used in conjunction with PCS.

7.1.3 The Send/Receive context switch bottleneck

During the implementation I discovered an interesting implementation issue. In systems that attempt to perform sends and receives with the same processor and without the help of a DMA, frequent context switches might occur due to simultaneous progress of a send with a receive.

This results from a particular problem with the extremely efficient fine grain communication on the iWarp. The data flow of the communication gates reach into the register file and the link level flow control affects reads and write from/to the register file. This mechanism was geared towards advanced streams in systolic algorithm, therefore flow control is always active and it might stall a read/write to the network interface at any time. It is possible but way too inefficient to guard every single gate register read operation. Therefore a message passing driver on the iWarp requires a two threaded RISC core to handle most flexible reads and writes by the processor. While the LANAI RISC core of the Myrinet network interface [14] has two execution contexts the iWarp processor core is a conventional sequential processor. Still there is some limited multi-threading functionality: The spools (DMA agents) can be seen as such form of multi-threading, because they can also make progress when a systolic gate (network access) stalls the main CPU.

Using a spool to send the message data out kills some flexibility for strided and indexed access pattern, but in turn, this allows the necessary multi-threading properties to make independent progress on sends and receive. For send operations that must be done by the processor an interrupt or a programmed task switch between the sender and receiver thread must take place. The minimal overhead and the communication efficiency for simultaneous send and receive critically depends on this overhead for context switching between send and receive activity.

In the iWarp architecture interrupts vectors (events) can be used to force a context switch between send and receive handler. The usable interrupt are: `MATCHED_ARRIVAL`, `SPOOL_COMPLETE`, `STOPCC_MARKER`, `REMOTE_QUEUE_EMPTY` or `BOTH_QUEUES_EMPTY`. To make most efficient use of those conditions the interrupt dispatch had to be reviewed. iWarp has 128 registers and 8 registers dedicated to interrupt handlers only. All memory is two cycle static RAM and there are no data caches. Therefore a context switch to an interrupt could be extremely light weight. Unfortunately there is an I cache with with a very particular 4-way associative structure featuring large code blocks as a cache line. It takes forever to load an I-cache line. An iWarp run time system must therefor dispatch most directly and with less than 3 jumps to make sure that no I-cache evictions take place. I-cache evictions could be entirely avoided with an iWarp component if the handler would be burned into the on-chip ROM memory of each node processor since instruction fetches from on-chip ROM are instantaneous and bypass the I-cache. Given the software development effort required to get the handlers right and ready the code only be included into the ROM of the forth or fifth release (D or E-step) of the iWarp component - and given that iWarp did not sell as well as Pentium it never made it to such a high stepping. Eliminating I-cache eviction remained impossible for the standard DMSG library and so the total overhead for processing the send and the receive of single a message with 2 interrupt dispatches an a few dozen lines of assembly code remained at 400 cycles or 20 microseconds.

To bring the per message overhead down I implemented a sender buffered alternative to the DMSG message passing system. The sender buffered driver uses a polling receive and uses interrupts at the sender side to inject message headers and trailers into the network. It uses four spools and dedicated injection queues to push messages quickly into the networks. With dedicated injection queues it saves the work to allocating a free queue (RESERVE_RPI) and is therfore more efficient. The message trailers are inserted upon a (SPOOL_COMPLETE) event. In the classic direct deposit implementation two interrupts are required to receive a message, while the sender works with polling. Conversely in the sender buffered library two interrupts must be dispatched to send a message while the receiver is completely based on polling.

With clever assembly programming and a systematic reduction of the context switching overheads for interrupt dispatch (in short - with lots of hacking) I-cache evictions could be eliminated and the code could be simplified to a few lines of assembly. The resulting overheads to process a message in sender buffered deposit messaging came down to about 125 cycles or 6.25 microseconds.

Unfortunately the sender buffered implementation of deposit message passing fell victim to a hardware problem with unreliable flow control in multi-lane routers. A lost flow control credit could result in an inconsistent communication state that could only be corrected by a global reset. Despite its nice performance the sender buffered deposit implementation never made it to a production code.

7.2 T3D implementation of direct deposit

The Cray T3D machines became operational in 1993. They are perfectly suited for the direct deposit model. The hardware can do remote store to a global address space with an extremely efficiency while the fetches remained a significant factor slower and the explicit handling of a FIFO queue made the cumbersome to program. The data transfer performance of 70 MByte/s per node for send and receive operation unique at that time and the node architecture is still highly competitive in todays world dominated by Beowulf PC clusters - at least when comparing just performance regardless of cost or price/performance ratios.

7.2.1 Hardware support

The hardware support with the Annex and the deposit/fetch engine is described in Section 2.1 of Chapter 2. The most important property of the T3D hardware support for global memory is that the mapping of remote address space to a local node is limited to a few communication partners. It is not possible to issue deposits or fetches to all nodes in the machines at the same time. A communication partner must be selected and the selection requires that all previous communication to the old partner is complete. Also there is a significant overhead for selecting a new communication partner. For these reasons the Cray T3D (and its successor, the Cray T3E) are classified as highly advanced message passing architectures.

7.2.2 Using SHMEM_PUT and SHMEM_GET

The primitives of SHMEM_PUT and SHMEM_GET transfer data from the user space of one processor to the user space of a communication partner. Their Syntax is:

```
void shmem_put (long * to_addr, long * from_addr, int length,
                int pe_num);
void shmem_get (long * to_addr, long * from_addr, int length,
                int pe_num);

     input:  a0      to_addr (remote PE)
             a1      from_addr (local PE)
             a2      length (in words)
```

```
            a3      pe_num
     output: none
```

The SHMEM_PUT and SHMEM_GET primitives can be used as send_deposit and receive_fetch operations for contiguous blocks of data. A later release of the low-level library also provided SHMEM_IPUT and SHMEM_IGET dealing with strided access to 64bit data. The library can not handle strided blocks of data or indexed data.

As indicated in the deposit- and fetch models there is no synchronization operation linked to the primitives. Synchronization is performed independently by lock or preferably by barriers in global communication operations. The SHMEM_BARRIER primitives uses the fast hardware barriers of a T3D system and is capable of synchronizing the entire machine in less than a microsecond. The library function combines the global barrier function with a local coherency operation guaranteeing that all remote operation are committed i.e. all remote stores are visible in the remote memory and all results of remote fetches have returned. Although the deposit model is used like *fire-and-forget* the hardware keeps track of completed operation and is able to use this information at synchronization points.

The SHMEM_PUT and SHMEM_GET primitives alone do not automatically guarantee the consistency of the cache in the microprocessor with the incoming data in the global memory. There is a choice between a conservative invalidate everything strategy that hammers the direct mapped cache with eager invalidates of all congruent cache lines or there is much weaker coherency strategy that invalidates cache only at certain synchronization points. In the first strategy the computation is slowed down by an incoming transfer invalidating many cache lines at random, while in the second option a relatively high constant overhead must be paid at each synchronization point. The Cray machine provides blazingly fast global memory access for data transfer, but leaves the worries about coherency to the compiler or the programmer. The support can be used to implement a wide variety of message passing model including my direct deposit model or the advanced buffer management scheme of fast messages implemented at UIUC [84]. Still because of its lack of global cache coherence, the T3D can not be programmed as a conventional shared memory machine.

7.2.3 Splitting SHMEM_PUT

The SHMEM_PUT primitive comprises two parts. A initialization part that establishes a mapping and a communication channel between a node and a communication partner and a highly optimize memory copy loop that actually transfers the bulk of data between the two nodes. The initialization section, called SHMEM_INIT looks as follows:

```
#define PUT_ANNEX 22
#define CACHED_READ 0x4 << 12
        .external _shmem_cpy_small
        .external _asm_annex_get_entry
        .external _MPP_PARTITION
shmem_init:
        ble     a2,return0      ; return if negative or zero count
        la      t4,_MPP_PARTITION  ; contains partition type
        ldq     t4,0(t4)        ; load partition type
        bne     t4,hw_part      ; 1 => hardware partition
os_part:                        ; Do PAL call.
        bis     a3,a3,t3        ; PE number into t3
        lda     t4,0(zero)       ; function code zero into t4
        bis     ra,ra,t5         ; save current ra in t5
        bsr     ra,_asm_annex_get_entry
        bis     t5,t5,ra         ; restore current ra
        bis     a0,r0,r0         ; set annex number in to_addr
        br      zero,return0
hw_part:                         ; Manipulate ANNNEX directly.
        lda     t4,PUT_ANNEX(zero)  ; annex number for remote store
        sll     t4,32,t4         ; shift annex number into position
        bis     a0,t4,a0         ; set annex number in to_addr
        bic     a0,^x1f,t6       ; get cache-aligned address for stq_c
        la      t1,CACHED_READ   ; set function code for cached read
        bis     t1,a3,a3         ; set function code and pe number
        stq_c   a3,0(t6)         ; set annex entry
        bis     a0,zero,r0       ; set annex number in to_addr
        br      zero,return0
return0:
        ret     zero,(ra)        ; return from here only if length <= 0
```

The call returns a pointer that points to the to mapped destination address in the communication partner. Typically a SHMEM_INIT call is followed by a simple load/store copy loop. The reader should not be fooled by the small number of RISC instruction in the code since some operations might activate PAL microcode and the entire sequence was measured to take about about a microsecond for the switch of a communication partner.

7.2.4 Building direct deposit into the FX compiler

Unlike in the iWarp case the T3 direct deposit system can perform a zero copy transfer to load strided data from its original location in the user data structure to the destination in the user data structure of the other node.

The Cray T3 direct deposit templates for the Catacomb backend of the Fx compiler use the packing routine do the actual data transfers and leave the actual templates for a blocked send and receive empty. The first part set maps the address space of the remote node and initializes the annex for remote communication. The second part is a macro calculating the array slices for a transfer and the third code fragment performs the communication for an array assignment statement b:=a in Fortran FX.

```
void ccom_dcomord(pproc,idx)
int *pproc, idx;
{
   double *a,*b;
   *pproc=(ccom_cellid+idx) % ccom_numprocs;
   ccom_dstproc= *pproc;
   a= (double *) &ccom_cellid;
   b= (double *) shmem_init(&ccom_cellid,&ccom_cellid,1,*pproc);
   ccom_remoffs = (double *) ((long) b - (long) a);
   barrier();
}
```

```
#define DPACKADP16_8(nonlocal,buf,bufptr,firstaddr,firstdata,\
                     numiters,addrmult,datamult) \
{ \
  int i; \
  double *a,*d; \
  a = (double *) ((long) firstaddr + (long) ccom_remoffs); \
  d = (double *) firstdata; \
  for (i=0;i<numiters;i++) \
    { \
      a[0] = d[0]; \
      a[1] = d[1]; \
      a = (double *) ((long) a + addrmult); \
      d = (double *) ((long) d + datamult); \
    } \
}

void CCOM_FUN_0(fx_a, fx_b)
  dcomplex fx_a[16][1024];
  dcomplex fx_b[16][1024];
{
  int await;
  ...
  rcellid = getcellid();
  scellid = rcellid;
  ccom_sync();
  await = 0;
  for (i=0;i<64;i++) {
    ccom_dcomord(&proc,i);
    lmslast_0=MIN((1023-(scellid*2))%128,1)+14;
    pu_0=(scellid*2)+1;
    pu_1=(proc*2)+1;
    for (outer_0=scellid*2;outer_0<=pu_0;outer_0++) {
      sfirst_0=(scellid*2)+PMOD(outer_0-(scellid*2),128);
      iprime_0 = outer_0-(scellid*2);
      lmsfirst_0=(((sfirst_0-(scellid*2))-iprime_0)/64)+iprime_0;
      rfirst_0 =(scellid*2)+PMOD(outer_0-(scellid*2),128);
      for ((pack0 =lmsfirst_0),(rslice0=rfirst_0);pack0<=lmslast_0;
           (pack0=(pack0+2)),(rslice0=(rslice0+128))) {
        for (outer_1=proc * 2;outer_1<=pu_1;outer_1++) {
          sfirst_1=(proc*2)+PMOD(outer_1-(proc*2),128);
          t_1=PMOD(outer_1-(proc*2),128);
          rfirst_1=(proc*2)+PMOD(outer_1-(proc*2),128);
          lmrfirst_1=(((rfirst_1-(proc*2))-t_1)/64)+t_1;
          DPACKADP16_8(proc-scellid,sbuf0,sbufptr,
                       &fx_a[lmrfirst_1][rslice0],
                        &fx_b[pack0][sfirst_1],
                        ((1023-sfirst_1)/128)+1,32768,2048);
      } } }
      if (proc != scellid) await = await + 1;
  }
  await_msgs(await);
}
```

A performance evaluation of such a sample array distribution statement is given in Chapter 9 and includes a comparison to hand coded direct deposit code as well as the corresponding transpose operation of a vendor supplied CRAFT parallel Fortran compiler.

7.3 Paragon implementation of direct deposit

Starting 1994 an implementation of direct deposit was attempted for the two processor Intel Paragon machine. Unlike the Cray T3D the Paragons were conceived as pure message passing machines and did not have any support for fine grain communication or remote memory in a global address space. Bulk transfers to and from the network interface necessarily involve some DMA agents. Therfore direct deposit must be implemented using handlers much like on the Intel iWarp systems.

For a simple implementation the SUNMOS [74] Operating System of Sandia National Labs was preferred over production OS derived from MACH and OSF1 since the SUNMOS runtime system did not involve paged virtual memory and permitted a flexible allocation of processor and co-processor to the various computation and communication tasks. The handlers were written

7.3.1 Twin Peaks implementation of direct deposit

As a consequence to the send/receive context switching overhead I devised a novel software structure for sending and receiving messages on the Intel Paragon two processor systems. The classic Paragon Software used one processor for computation and one processor for communication. The SUNMOS operating system of Sandia National Labs allowed to use both processor for computation, while one processor dealt with communication on a part time basis. Twin peaks uses both processors for computation and both processors for communication if needed. To make context switches infrequent and cheaper the two processors are specialized, one processor performs all communication operations dealing with sends and the other processor performs everything dealing with receives. As indicated in the section on the iWarp implementation send and receive could not block each other and progress must be made in both activities.

The differences in the design can be seen in the state diagram for the handler dealing with sends and receives of messages. Due to the limited hardware capabilites and the several bugs or conceptional glitches the state diagram are highly complex. The first diagram in Figure 7.1 shows the classic SUNMOS communication handler that use only one processor. It can be seen that there are many state transitions between send and receive states. The second diagram in Figure 7.2 shows the twin peaks implementation where one processor handles a state machine for sending messages and one processor handles a state machine for receiving messages.

Based on the state diagrams and my experience with the iWarp handlers I can argue that a specialization of processors to specific tasks results in smaller code complexity, a smaller footprint and a higher performance of a message passing system. Measurements on overheads for message transmission and reception seem to confirm. This contradicts some findings based on simulation that looked at SMP mul-

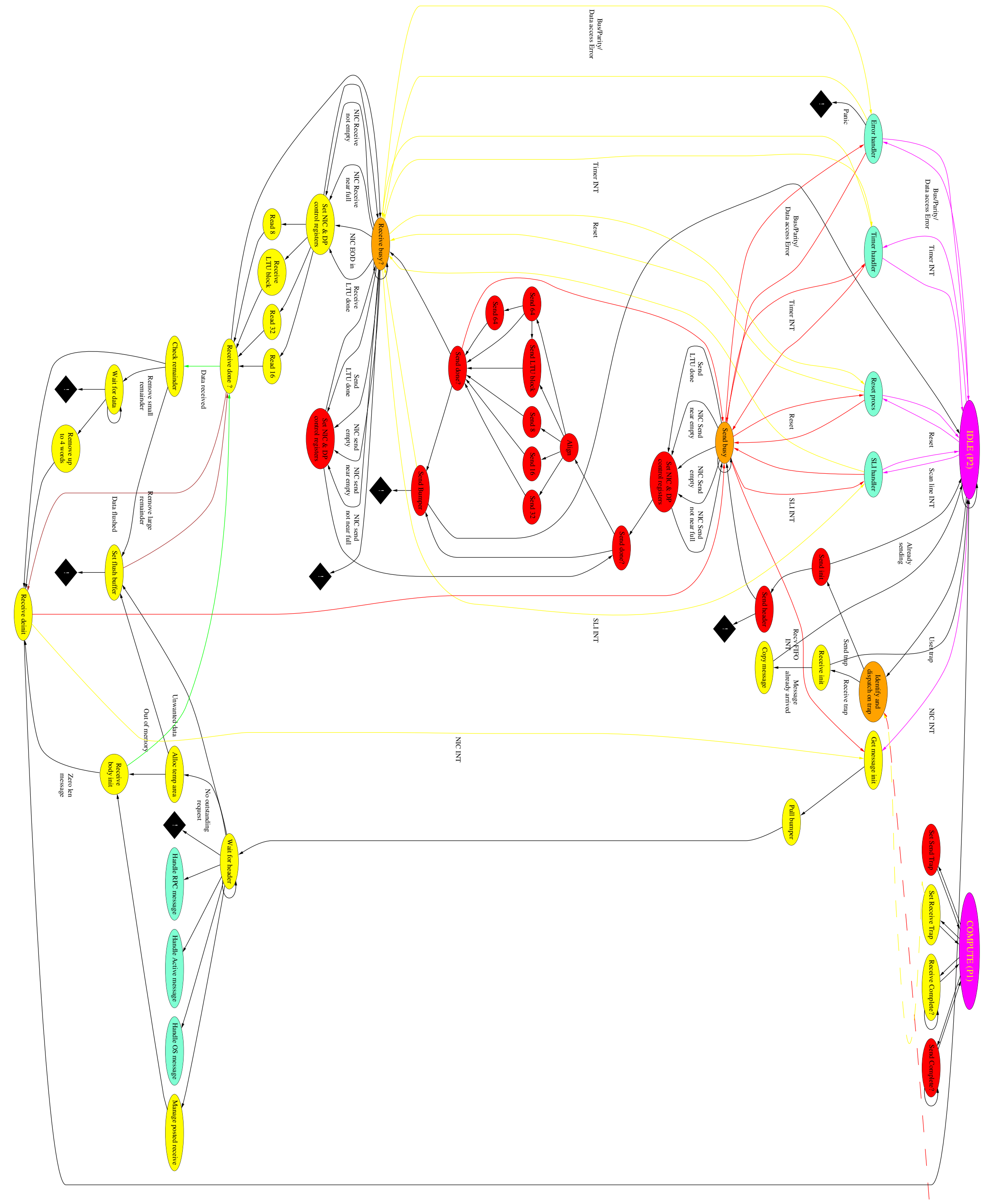


Figure 7.1: State diagram of handler according to the sunmos philosophy.

tiprocessors and observed that a flexible allocation of processors for message handling by the SMP as a whole entity performed better than a separate message handling by each processor [34].

Myself and CMU technical staff spent more than a person-year in Paragon related development and

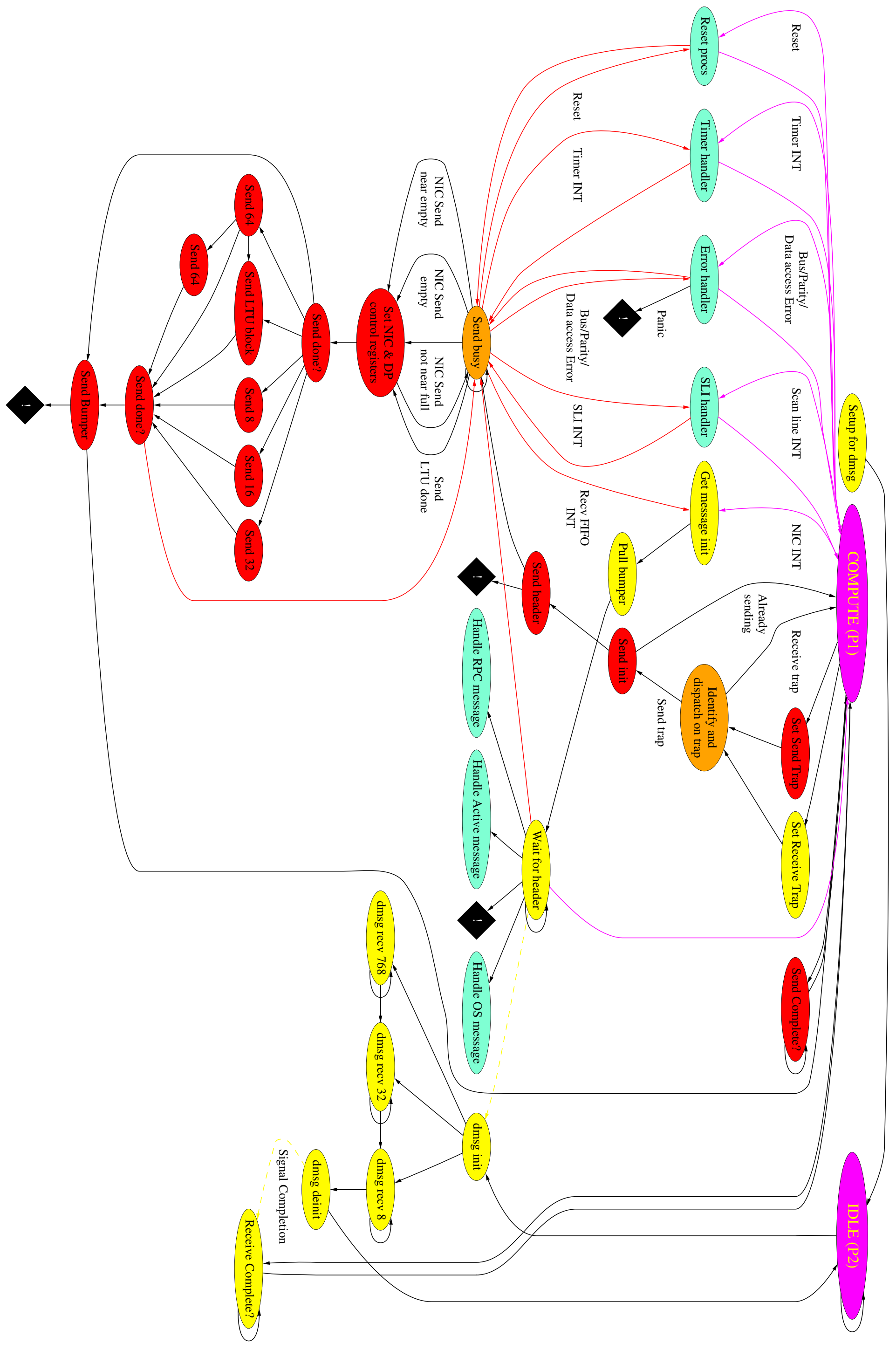


Figure 7.2: State diagram of handler according to the twin-peaks philosophy.

we obtained at least the most basic performance numbers for direct deposit messaging with a twin peaks implementation. The numbers are the basis for the considerations in the subsequent sections on memory system characterization and performance evaluation.

Despite a promising concept and many interesting issued involved the twin peaks direct deposit message passing system never made it to a production release. There were too many glitches in the hardware to get all things working correctly. In particular there exist about 1001 special cases, when the network interface status bits were found lying to the programmer. In some cases there are indication that a double word is in the FIFO, while actually no data is available. This problem could also generate a freeze of the DMA that was very hard to remote. The production OS used a multi second timeout followed by a cleanup and retransmission of the message for such a case. A more efficient zero copy messaging system like direct deposit is left without a chance to work correctly on such a hardware platform. It is disturbing to see that the vertically integrated of effort of hardware and software design team is not longer compatible with the business model of todays computing industry. Hardware is developed in short cycles and sold regardless as soon as it works almost correctly. It is not relevant if the software companies higher up in the developer chain do have problems using it or not. As for high performance massively parallel machines the Cray T3D and T3E mark the end of an engineering tradition.

7.4 DEC8400 implementation of direct deposit

An implementation of direct deposit on a fully cache coherent shared memory machine seems quite easy. The POSIX packages for shared memory can be used to:

1. Fork n threads on a n processor machine.
2. Bind each thread to a specific CPU.
   `bind_to_cpu(pid,1<<cidl,BIND_NO_INHERIT);`
3. Unmap a certain amount of address space.
   `munmap(baseaddr,ccom_memsize)`
4. Attach a home memory segment to each processor.
   `ccom_homeseg=shmat(ccom_shmid[cidl],baseaddr,0)`
5. Map the home segments of all other processor into the address space of all other processors.
   `for (m=0; m<ccom_numprocs; m++)`
   `ccom_memseg[m]=shmat(ccom_shmid[m],0,0))`
6. Use copy transfer loops to move data between the home memory segment.
7. Allocate POSIX semaphores for synchronization between the processes.
   `ccom_semid=semget(IPC_PRIVATE,ccom_numprocs,IPC_CREAT | 0666))==-1)`
8. Build a global barrier with the semaphores.
   A combination of `rc=semop(s,&sem_ds,1)).`

With these programming tricks a software environment for global remote memory can be created although the has fully cache coherent global memory and could offer a much more comfortable programming model. For my investigation I was interested in a reference implementation of the Catacomb compiler backend on the DEC 8400 and was also interested whether with explicit data movement it would be possible to move data more efficiently than with the default shared memory mechanisms.

In such a setting the parallelizing compiler works like on a distributed memory machine and explicitly moves data by copying it from one home memory segment to another. Such copies can use the best know code sequence to copy memory with different access patterns (e.g. strides or indexes). False sharing can be carefully avoided. Of course all home memory segment are stored in the multi back global memory system and a processor can not force a deposit into the caches of another CPU. There is simply no hardware support for that in a bus based symmetric multiprocessor. Still the writes of a deposit operation to the home segment of another processor will eventually reach the memory banks in the global memory system. Later - if another processor wants to use the data in its own home segment it is free to do so, but at the cost of pulling the data over to its caches. This happens automatically through the cache coherency protocols when the data is accessed and is as close one can get to the deposit model for global memory operations.

I used this auxiliary construction of "non-coherent" remote memory on the DEC 8400 for a memory system characterization and for a most compatible port of parallelizing Fx compiler with its direct deposit backend to a symmetric shared memory machine. Most high performance computing users would probably switch to shared memory codes on SMPs rather than use a HPF code on an SMP or go for a different compiler backend to target the SMPs directly. Still the experiment with direct deposit Fx Fortran on a genuine SMP was probably highly useful to obtain performance numbers for a comparison.

The limitation of this approach are in the Operating System data structures that might not permit to partition and map the entire memory as shared memory segment. I ran into such limitation and were only able to use a fraction of the 4GB of main memory of the four processor DEC 8400 installed at the Pittsburgh Supercomputer Center.

7.5 Released Software

The DMSG Communication Library for iWarp

The Deposit Message Passing Communication Library for the iWarp was released in 1992 by the David OH́allaron, Jim Stichnoth and the author.

A fast array transposes package at the PSC

The CMU deposit-model for decoupled synchronization and copy-transfer model for optimized memory operations provide the optimization techniques that result in a highly competitive library routine for fast array transposes (libtp.a).

The library was released to the T3D/CRAFT engineering team of Cray Inc. and to the user of the Pittsburgh Supercomputer Center.

The following code segments contain common declaration and some pseudocode for the direct deposit transpose routines. The real routines contain a number of memory system optimization such as unrolled loops, folded constants etc...

```
Declarations for Direct Deposit and PVM

#define P  256

#define NM 4096+4
#define N 4096
#define RM (NM/P)
#define R (N/P)

#pragma _CRI cache_align a,b

static doublecomplex a[RM][NM];
static double fill[4];
static doublecomplex b[RM][NM];
```

```
Pseudo code for direct deposit transpose

i0=my_cell_id;
for (j=0; j<P; j++) {
  j0=comm_schedule[j];
  barrier();
             /* congestion control */
  b0=shmem_map(b,a,1,j0);
  for (l=0; l<R; l++) {
    for (m=0; m<R; m++) {
             /* copy loop w. remote stores */
      b0[m][i0*R+l].r=a[l][j0*R+m].r;
      b0[m][i0*R+l].i=a[l][j0*R+m].i;
    }
  }
}
barrier();
shmem_udcflush();
             /* re-establish cache consistency */
```

Comparison to vendor transposes

Most users of the Cray T3D program transposes with PVM the message passing communication library provided by the vendor.

Pseudo code for PVM3.3 transpose

```
for (j=0; j<P; j++) {                /* pack buffers */
  for (l=0; l<R; l++) {
    for (m=0; m<R; m++) {
      af[j][m*R+l].r=a[l][j*R+m].r;
      af[j][m*R+l].i=a[l][j*R+m].i;
} } }
for (j=0; j<P; j++) {                /* do sends */
  pvm_initsend(0);
  pvm_pkdouble(af[j],R*R*2,1);
  pvm_send(j,0);
}
for (j=0; j<P; j++) {                /* do receives */
  pvm_recv(j,0);
  pvm_upkdouble(bf[j],R*R*2,1);
}
for (j=0; j<P; j++) {                /* unpack buffers */
  for (m=0; m<R; m++) {
    for (l=0; l<R; l++) {
      a[m][j*R+l].r=bf[j][m*R+l].r;
      a[m][j*R+l].i=bf[j][m*R+l].i;
} } }
```

CRAFT, common declarations:

```
      real a(N,N)
      real b(N,N)
      intrinsic transpose
cdir$ cache_align a,b
cdir$ shared a(:,:block)
cdir$ shared b(:,:block)
```

CRAFT work-sharing transpose:

```
      do i = 1,N
cdir$ doshared (j) on b(i,j)
         do j = 1,N
            b(i,j)=a(j,i)
         enddo
      enddo
```

CRAFT intrinsic transpose:

```
      b= TRANSPOSE(a)
```

The performance scalability curves in Figure 7.3 outline the per processor communication performance achieved on transposes on the Cray T3D. The transpose is for a 2-dimensional array with double complexes as elements, the problem size is 2048x2048. The direct deposit model starts at 40 MByte/sec with 2 processors and scales gradually down to 22 Mbyte/sec on 512 processors as more processors get involved and the messages become smaller. The transpose intrinsic of CRAFT (Cray Parallel FORTRAN) is certainly programmed in a similar way but must rely on at least one unnecessary internal copy since its transfer rates are about half of direct deposit. The PVM implementation has gather, scatter into contiguous blocks and other additional copies. Furthermore the constant overhead per message makes PVM communication completely impractical for small message sizes of 2k by 2k transposes on larger machines. The work-sharing implementation of CRAFT misses to detect the transpose as global regular operation and issues extremely inefficient code for pulling the data over element by element.

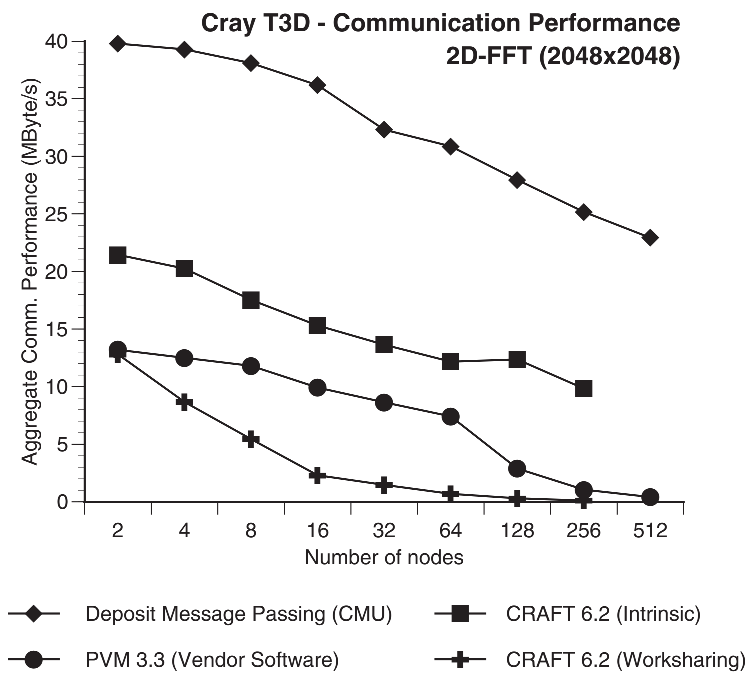


Figure 7.3: Per processor communication performance of a 2048×2048 point 2D-FFT for different machine sizes of Cray T3D.

The second graph in Figure 7.4 shows how communication performance affects scalability of the transpose operations. The direct deposit and the CRAFT intrinsic transpose scale fine up to 256 or even 512 nodes. The PVM performance falls apart above 32 or 64 processors, while work-sharing does not scale far beyond 8 processor. Note that the curve is a log-log scale and that the quarter inch between the two curves means a factor of two performance difference.

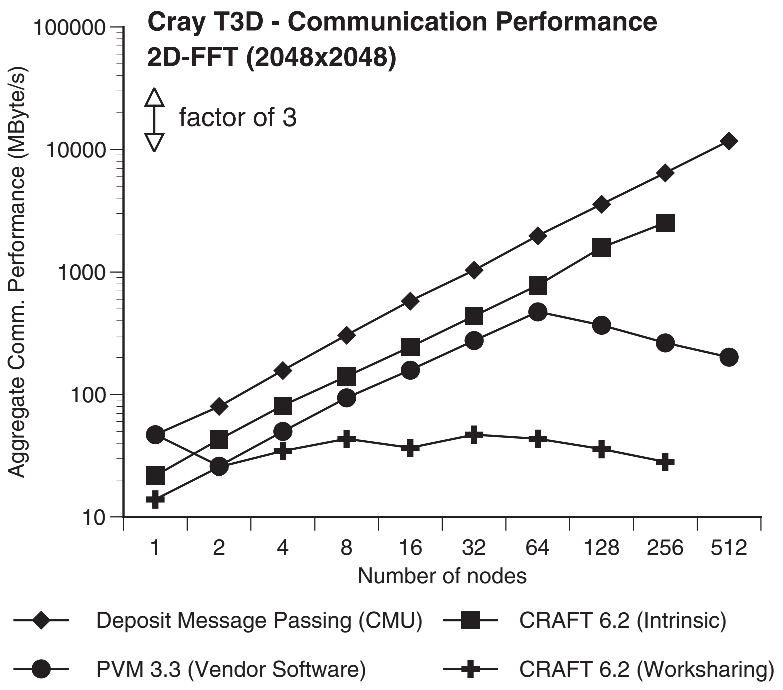


Figure 7.4: Aggregate communication performance of a 2048×2048 point 2D-FFT for different machine sizes of Cray T3D, graphed on a log-log scale.

The survey of PVM and CRAFT indicates that at the time of the measurements the transpose based on direct deposit was by a factor of two faster than the best transpose routine by the vendor. The CMU transposes package is installed at the Pittsburgh Supercomputer Center is available to all Cray T3D users.

Chapter 8

Memory System Performance Characterization

For many applications, it is the memory system of a computer that determines their performance. The FLOPS (millions of floating point operations per second) rating of a system may be impressive, but if the memory system cannot supply operands at the right speed, application performance may be quite disappointing. This topic is especially important for high-end, parallel systems, which contain multiple processors to boost peak computation-only performance.

Many parallel systems provide the model of uniform memory access (all memory cells can be addressed by any processor); however the term "uniformity" applies only to the access model, not to performance. The actual performance of the memory system depends on parameters such as the working set size, the stride, and whether the processor attempts to read/write local or remote data. To obtain the best performance for applications, the compiler writer or application developer must understand the performance implications of different types of memory accesses. Whereas private-memory machines (e.g. a network of workstations) force a programmer or compiler to optimize the program for efficient memory access just to obtain correctness, shared-memory systems (providing a uniform view of memory) do not require such an optimization. But to obtain respectable performance, it is also advisable on shared-memory machines to move data close to the processor that operates on them. So communication (data transfers between different parts of the machine) is an important part of many parallel programs.

The designers of memory systems have a number of options to bridge the performance differences between local and remote memory accesses. The design of a memory hierarchy is a well-known strategy, and recent processors include on-chip caches to bring the lower levels of the memory hierarchy closer to the processor. However, even with the presence of on-chip caches, there are many design options left. To get an understanding of how such memory systems behave in practice, I investigate the memory system performance of five parallel systems.

The Intel iWarp has a relatively primitive, but very powerful memory system. The excellent performance relative to the processor is given by the SRAM-based architecture and the low clock rate of this older architecture (systems on the market by the end of 1990). The memory system design offers a uniform performance at L2 cache speeds across the whole address space. Unfortunately, SRAM memory is extremely costly, and the total amount of main memory available on each iWarp node is also just about the size of an L2 cache in more recent systems.

The Cray T3D, the Cray T3E, and the DEC 8400 are three recent systems that have some features in common, yet include completely different memory systems. The DEC 8400 is a shared memory, symmetric multiprocessor based on a high-speed bus. The Cray T3D and T3E are scalable multiprocessors based on a 3D torus interconnect. All systems have in common that (i) they are based on the same processor architecture (the DEC Alpha), (ii) each processor can access the complete memory system (i.e., all memory locations have a unique address that can be used by any processor), and (iii) each processor contains at least a small on-chip cache. However, the T3D uses a different implementation of the Alpha, and there exists a big difference in the access mechanisms: with the Cray T3D and T3E, the processor distinguishes between local and remote memory, and the L1/L2 caches of different processing elements are non-coherent among each other. Locally the caches may or may not be coherent with local memory or the communication interface handling remote accesses (both machines allow selection of the invalidate policies at load time of a program). Consequently, either a sophisticated compiler or a highly skilled programmer is required to map an application onto this system. In the case of the DEC 8400, all memory locations are accessed using the same instructions, and the bus-based hardware maintains coherence of a global multi-level memory hierarchy with memory banks and caches. This organization puts the burden of maintaining coherency on the hardware and simplifies the programmer's or compiler's task. Furthermore, the DEC 8400 employs an off-chip L3 cache, whereas the Cray systems instead employ a streaming unit to improve memory bandwidth for strided accesses.

The memory system of the Intel Paragon is somewhat of a hybrid between a simple, cache-based memory hierarchy and the advanced stream-based solution used in the Cray T3E, the most recent machine. An on-chip L1 cache adds the characteristics of a multi-level hierarchy and the pipelined loads of the i860 instruction set contribute the streams support to this memory hierarchy.

Given the complexity of the memory system design of modern high-performance systems, and the absence of a simple analytical model, compiler and application developers need guidance on how to implement the basic data transfers. In this chapter I put the extended copy transfer model to work and show how its key performance parameters can be combined to obtain a realistic characterization of memory system performance. The experimental framework (and the environments that may apply such performance data) are sketched in Section 8.1 in this chapter. The key differences in the memory designs of these systems are presented in Section 8.2. Then I discuss performance for local and remote accesses in Sections 8.3 and 8.4. Section 9.3 in the chapter on evaluation finally extends the discussion from micro-benchmarks to a complete application kernel.

8.1 Experimental setup

I empirically evaluate the local and remote memory system performance of five parallel systems. Among the systems I have two Intel systems, and three systems are based on implementations of the DEC Alpha processor architecture. The Intel systems are the iWarp, based on a 20 MHz full custom processor, and the Paragon, based on a 40 MHz i860XP off-the-shelf processor. The iWarp was developed in a joint project between Intel, CMU, SPAWAR and DARPA. The three DEC Alpha based systems are the DEC 8400 (300 MHz 21164, alias EV-5), the Cray T3D (150 MHz 21064 alias EV-4), and the Cray T3E (300 MHz 21164 alias EV-5). The comparison of the three DEC Alpha systems reduced the architectural difference to a minimum and allows for a maximum of comparisons.

The five systems cover a wide spectrum, from a fully cache-coherent shared-memory architecture to completely message-oriented distributed memory architectures. The DEC 8400 is a cache coherent, shared memory, symmetric multiprocessor (SMP) based on a high-speed bus offering sequential consistency; I used a four-processor system and also repeated some measurements on an eight-processor system. Both Intel systems (Paragon and iWarp) are distributed memory parallel computers designed for message-based communication. The iWarp supports systolic, stream-based communication in addition to the message-based communication. The Intel systems use a 2D topology: the iWarp systems at CMU are configured as square, 8×8 tori and our Paragon contains a rectangular mesh surface of 6×11 nodes. The Cray T3D and T3E are scalable multicomputers based on 3D torus high-speed interconnects and support remote memory access in hardware, but they leave global memory consistency to the application or compiler. For the Cray T3D measurements I used a four-processor partition out of a system with 512 processors. For the Cray T3E data I relied on a small-scale development test-vehicle currently installed at the PSC to assist the port of existing codes, and on an engineering prototype system at Cray Research for selected measurements; the installation of a full-scale T3E production system is anticipated in time for the final paper.

For readers not familiar with the architectures of the five systems, I describe the important details relevant to the memory and communication system interface in Section 8.2. Information about other differences is available from many sources: DEC 8400 [29, 30, 39, 31], Cray T3D [29, 1, 9], Cray T3E [90, 19]. In addition to the technical reference material of the vendors, other research groups evaluated some aspects of these machines [5, 62, 98]. Recently, an empirical study comparing the two Alpha processors based on standard benchmarks provides useful insights using performance metrics not related to the memory system, like instruction issue, branch prediction and CPI figures [26].

My goal is to compare the memory systems of these modern parallel systems. Since it is nowadays usual to use a compiler to map applications onto a parallel system, I are particularity interested in an investigation that pays special attention to the access patterns encountered in compiled code. A good description of the typical workloads for this class of machines at the Los Alamos and Lawrence Livermore National Laboratories was given in [110]. In particular, the authors argue for the practical importance of vectorizable memory-intensive workloads and point out the difficulties of adapting such workloads to the cache-oriented memory systems of microprocessors.

8.1.1 Parallel machines as compiler target

This investigation of memory system performance is part of the Fx Fortran compiler project; Fx [101] is a dialect of High Performance Fortran (HPF) [67], and the Fx compiler has been re-targeted to a number of parallel systems, including the aforementioned five systems. For many applications, the key to getting good performance is to generate efficient communication code; communication code is any code that moves data from one memory zone to another. For a private-memory system (like the Paragon, iWarp or network of workstations), communication code may involve explicit "send/receive" operations and is required for correctness; for a shared-memory system like the DEC 8400, such code may involve extra memory copy operations that may be advantageous to improve performance (but may not be required for correctness). On SMPs like the DEC 8400, I call a transfer *remote* if one processor writes data to memory and another processor reads it, and *local* if the same processor reads and writes the data.

To tune the performance of the Fx compiler, I first measure the basic performance for key operations of the "copy-transfer-model" to obtain performance figures for local and remote transfers. The characterization produced by those micro-benchmarks allow the communication compiler to pick the least expensive way to move data in the system.

8.2 Differences in memory system design

I now give the key specification of the microprocessors and discuss some key aspects of the memory system design of the five machines. The parallel machines used for this memory characterization are members of different generations, since their product lines appeared on the market on different dates between 1990 and 1996. The resulting spread in processor cycle (about a factor of 15) must be kept in mind when comparing the absolute memory system performance, although the numbers are normalized to MByte/s and are always taken with 64-bit quantities. But despite this big difference in clock rates, the overall memory system performance remains surprisingly comparable. The widely different characteristics for different kinds of access patterns and different working sets makes an in-depth comparison of memory systems interesting and worthwhile.

8.2.1 DEC 8400

The memory system design of a node in the DEC 8400 symmetric, shared multiprocessor is based on a 3-level hierarchy of caches (L1, L2, L3) on the processor board and a shared dynamic memory (DRAM) on separate boards. The first two levels of caches are integrated on the DEC Alpha 21164 processor chip. The L1 cache is just 8KB in size; it is a simple, data-only, direct mapped, write through cache, but the access latency is only two clocks (i.e. 6.6ns). Furthermore, this cache provides an extremely high bandwidth (4.8 GByte/s in theory). The L2 on-chip cache is a 96KB 3-way associative unified instruction/data cache; it has a write-back latency of 6 clocks and a peak bandwidth of 2.4 GByte/s. The write-back L3 cache is built out of SRAM.

The DRAM memory of a DEC 8400 is built from memory modules, which are two-way interleaved. With four memory modules, a maximal interleaving of 8 can be reached. Like a workstation, the DEC 8400 supports virtual memory. The vendor lists 176ns–928ns as an average latency for load operations from main memory, depending on how many memory modules (i.e., memory banks) are installed and how many processors compete for memory access. The system used for these measurements contains four memory modules with a total of 4GByte. For large contiguous transfers, the DRAM memory is faster than the L3 caches of another processor; this fact indicates that the memory system includes modest stream support for large contiguous transfers.

In a symmetric multiprocessor both memory accesses and inter-processor data transfers involve the bus-based communication system. The DEC 8400 is built around a high speed system bus with 40-bit address and 256-bit data path. In the DEC 8400, the bus is clocked at 75 MHz, a quarter of the clock frequency of the microprocessor, yielding a peak transfer-rate of 2.4 GByte/s across the system bus. This limit is reduced to a peak of 1.6 GByte/s under the best burst transfer protocol [39, 31]. A bus system puts limitations on scalability (fixed number of slots for processor and memory cards) but provides free broadcast, simplifying the implementation of globally coherent caches significantly.

8.2.2 Cray T3D

In the Cray T3D, the caches play a much smaller role than in the DEC 8400. The processing node of the T3D consists of a DEC 21064 processor with just an on-chip L1 cache; the memory system includes a DRAM-based memory system on the same board, which is interfaced with fast ECL external support circuitry to the processor as well as to the communication system. The on-chip L1 cache is 8KB in size, data-only, direct mapped and write-through, read-allocate. The write path contains an on-chip write-back queue that buffers the high rate processor writes and coalesces them into 32-byte entities if they are contiguous. The external circuitry supports contiguous reads with a read-ahead logic that can be turned on/off at program load time. With its completely different read and write paths and the external read-ahead logic, this memory system supports streamed access to large amounts of data. DRAM accesses within the same DRAM page are accelerated; see the technical data sheets or [1] for details regarding different speeds and bandwidths.

Remote accesses are performed to a network interface, which is also built from ECL gate arrays. Remote stores are directly captured from the write-back queues, while remote loads can be performed in a transparent blocking manner at minimal speed, or somewhat faster through an external FIFO pre-fetch queue located in the support circuitry. Simultaneous communication is limited to one (or a few) communication partners, and there is a "per-message" overhead for switching partners. The switching fabric is arranged as a 3D-torus of fast ($>$ 200MB/s) links.[1]

Every node has some fetch/deposit circuitry that handles incoming remote operations (loads and stores) with their memory accesses on behalf of the communication system. These accesses can happen without involvement by the processor at the receiver node (i.e., there is no requirement to generate an interrupt). This circuitry can store incoming data words directly into the user space of the processing element, since both address and data are sent over the network. The on-chip cache of the main processor can be invalidated line by line as data is stored into local memory or can be invalidated entirely when the program reaches a synchronization point.

8.2.3 Cray T3E

The design of the memory system of the Cray T3E is similar to the T3D in the sense that it too includes support circuitry for non-local operations, stream support, and carefully tuned DRAM performance. However, this memory system inherits a cache structure from its DEC 21164 processor. As for the DEC 8400, the first two levels of caches (L1, L2) are integrated, since the T3E is also based on the DEC Alpha 21164 processor chip. They have the same sizes, organization and latencies as in the DEC 8400. There is no L3 cache, but the memory system includes support for memory streams. At this time, I cannot include a complete discussion about the design of the streaming units and the pre-fetch registers. Some questions remain open, e.g. to what extent support for prefetching/streaming can be controlled by the program. I expect to resolve such questions by further experiments on the first production machine to be installed in inn the near future at the Pittsburgh Supercomputer Center and more details of its architecture become published.

The remote accesses are performed to a network interface in the support circuitry. Remote stores are directly captured from the write back queues, while remote loads are performed through a set of E-registers. For the moment, I rely on the shmem_iput and shmem_iget communication primitives provided by Cray Research for the Cray T3E. The communication network is similar to the Cray T3D but every processor has its own network access; consequently, the raw link throughput improves significantly over the T3D.

8.2.4 Intel Paragon

In the Intel Paragon the memory system comprises an on chip L1-cache, a DRAM-based memory system and two line transfer units (DMA controllers) in its basic processor node architecture. The internal caches in the i860 microprocessor are unified, 4-way associative and configurable in either write-back or write-through mode of operation. The paged memory mapping hardware (TLB) can be specialized for the memory management of the supercomputer by selecting a segmented mapping of 4MB pages rather than the common 4KByte pages used in workstations with demand paging. However neither the original Paragon operating system (OSF/1) nor the simplified third party SUNMOS operating system take advantage of these options. Both use small pages for the translation and configure the cache as write-back. In programs using large amounts of memory and strided accesses, TLB misses are frequent, but relatively cheap since they are handled entirely in hardware.

[1]The actual implementation pairs two processing nodes with a single network access. Therefore the effective link speed seen by each of the two processors falls back to 70 MByte/s, but the largest machine size that can route AAPC permutations without congestion increases to 1024 nodes.

The most interesting part of the i860XP microprocessor architecture is its instruction set with pipelined floating point loads that bypass the caches and provide a high speed streamed access to main memory. There is no pipelined store instruction and no write-back queue limiting the write-stream support to the write behind behavior caches. Write back caches are far less efficient than write back queues for coalescing stores, because they write-allocate entire cache lines — to the disadvantage of the programmer whenever strides become involved.

For remote accesses the Intel Paragon node relies either on the line transfer units (its DMAs) or the processor. Both can move data from the network interface to the memory system of a communicating node or vice versa. More precisely, a Paragon node architecture is a twin processor architecture with a second microprocessor sharing the memory system and the network access ports. I can dedicate this second processor to memory operations and treat it like an intelligent DMA if I like.

The memory system is linked to the processors by a medium speed intra-node bus with burst transfer capability of 400 MByte/s. Externally the nodes are linked by channels capable of moving data at 200 MByte/s between two network interfaces. Larger machines are configured as a two-dimensional surface mesh. The system installed at CMU has 6×11 nodes and runs on a 40 MHz clock. The "not to exceed" performance is 60 MFlops per node in double precision.

8.2.5 Intel iWarp

The iWarp memory system design is marked by the requirements of stream-based computation found in systolic algorithms. At the design time of this early distributed memory parallel machine (i.e., 1987-89), it seemed appropriate to build the memory system from fast static memories (SRAM) and thereby take a blanket exemption from all the problems related to memory systems for high speed microprocessors (or at least from problems other than maximal size and cost!). The iWarp memory hierarchy is flat and has no data caches (there is caching for the instruction stream). The standard iWarp memory system is built entirely from SRAM chips (25ns access time) and is capable of high bandwidth and low latency. The memory system is so fast that its data stream can feed multiple operands to the arithmetic units at full instruction issue rate. The port to memory is roughly 88 bits wide (64-bit data, 24-bit address), consuming around 100 pins of the 271 pins of the iWarp chip.

The uniform, low access latency and the "one per clock" load issue rate translates into impressive memory access bandwidth of 160 MByte/s peak from main memory — in theory — and up to 106 MByte/s in practice. The instruction set permits one double word load operation to occur with as little as 2 clocks (100ns) latency and at a sustained rate of one load completed per clock. This mode of operation requires the activation of a memory pipeline accessible only within 96-bit compute and access instructions. Under more practical boundary conditions, i.e., for loads with address generation through pre/post increments of registers the latency is increased to 3 clock and bandwidth reduced two loads per 3 clocks, resulting in the measurable value of 106 MByte/sec. The issue rate of stores is limited to one double word per 3 clocks. [2]

Incoming and outgoing remote accesses (memory-based) communication is handled by either a DMA (spooler) or the processor in a tight loop. The DMAs are capable of transferring a long word in 1 clock or a double word at a 25% duty cycle (each of the eight in-bound or out-bound communication channels can deliver a double word every 4 clocks). Alternatively the processor can transfer data at the same 4 cycle per word rate within a tight load/store loop, but at a 75% to 100% duty cycle. The unique support of systolic gates incorporates the network interface directly into the register file (gate register). This is as close as one can get to the network, and receive-store or load-send operations can be composed of loops with just

[2] The architecture is tuned for maximal performance with 32-bit words, i.e. 2 clocks latency and 2 words per 2 clocks sustained rate. However, for a comparison with the 64-bit DEC Alpha architectures, I concentrate on the performance with 64-bit words at this time.

a single load or store instruction between memory and a gate register (the instruction set has an endloop bit and the architecture supports zero cycle branches for loops).

8.3 Performance metrics and benchmarks

Several prototypes of early shared memory computers led to the definition of a model that characterized communication and local memory system performance with a single parameter, the access latency. This model is known as the non-uniform memory access model (NUMA) and a large number of papers like [113] analyze algorithm performance under that model. The NUMA model does not distinguish between local and remote memory and describes memory only as fast and slow memory in terms of access latency. The problem with the basic model is that it takes the reciprocal value of latency for bandwidth and fails to capture cases of pre-fetched data, pipelined transfers, or differences due to access patterns, i.e., contiguous, strided, and indexed accesses.

8.3.1 Memory and communication system bandwidth

For the communication and memory operations generated by a compiler for a data parallel language (e.g., HPF), the actual transfer bandwidth is more important than the access latency. Compilers like the Fx compiler have a good knowledge of the access pattern and are well able to use pipelining and arrange large transfers in an optimal manner. Often the only issue that counts is the rate at which a given amount of data can be moved for a particular case, and the latencies of a single memory access are irrelevant for most large-scale scientific or data-intensive applications.

In the *copy transfer* model of Chapter 6, each communication step is seen as a composition of basic copy transfers with known performance characteristics. If a given platform allows more than one way to implement a communication step, the modeled bandwidth metric is used to determine the best way to implement this communication step. Again I call memory accesses that go to local caches or to DRAM memory with the same processing node or processor board *local accesses*. Accesses that cause data to traverse a processor board boundary are local or remote accesses, depending on whether the reading processor (consumer) and the writing processor (producer) are the same or are different.

My investigation of transfer performance deemphasizes the different cache coherency models used in the symmetric multiprocessor (DEC 8400) and the distributed memory machines (T3D and T3E). The coherency models reflect only the difference between explicit and implicit communication in either a message passing or a shared memory machine. The coherency model might be responsible for differences in the instruction streams performing a given copy transfer, but not for the performance of the *most efficient* way to do a transfer. On the T3D and the T3E there is no support for global cache coherence; in some cases it is better to turn even local coherence (within a node) off for a particular transfer operation and establish consistency later at a synchronization point.

8.3.2 Micro-benchmarks for the memory hierarchy

A few small, but highly optimized, memory system benchmark programs were used to measure the memory copy bandwidth under strided memory access for different working sets. The same micro-benchmark programs are used on the DEC 8400, the Cray T3D and the T3E.

Two different basic memory operations are examined, both of which operate on 64-bit double words.

Load Sum A load operation and an add-summing operation. All elements of the working set are used to accumulate a sum.[3]

Load/Store copy All data of the working set is copied by either loading it with a fixed stride and storing it contiguously, or by loading it contiguously and storing it with a fixed stride. Such copy transfers are common in transpose operations.

A third “Store Constant” benchmark was written as a dual to the “Load Sum” benchmark to evaluate store performance. The resulting graphs did not add enough insight to the picture to warrant the space in a short conference paper. The store benchmarks confirmed the specified, default write-back policies of the caches and the proper function of the write back queues.

The Fx parallelizing Fortran compiler produces low-level C code for all interprocessor communication. Therefore, I generated my benchmarks with the vendor’s production C compiler rather than in assembly. However they are carefully crafted C routines, inspected for well-scheduled machine code.

On the DEC 8400 and the T3E node, a memory bandwidth of 2400 MByte/s corresponds to the delivery of one 64-bit operand (8 Bytes) per CPU clock cycle (300 MHz). On the T3D a peak of 1200 MByte/s could be achieved when one operand is delivered per clock cycle (150 MHz). Both CPUs are supposed to reach their peak bandwidth when accessing data out of L1 cache. Unfortunately, not even the vendors’ own compilers can generate the necessary instruction schedules for such a memory system benchmark. With a lot of careful C-code tuning and much hand-holding, I measured about half of the peak bandwidth for loads out of L1 cache with compiler-generated benchmarks. The other levels of the memory hierarchy are much slower and are not affected by this compiler problem.

Unless noted otherwise, the memory system benchmarks are executed on either a single processor of the shared memory machine (other processors idle), or on a single-node partition of the distributed memory system.

8.4 Characterization of the local and remote accesses

I use the two classes of micro-benchmarks to probe the multi-level memory hierarchy of the two machines under consideration. In particular, I test the support of the memory system for *temporal locality* and *spatial locality* separately. The performance of the memory system is measured in access bandwidth for different strides and different working sets. The stride parameter shows how well caches and external stream logic help with read ahead and other means of improving bandwidth for accesses with spatial locality. The working set parameter shows how the memory hierarchy supports temporal locality, i.e. the effect of cache hits through reuse of recently accessed data. The micro-benchmarks access all locations of the working set exactly once, but start with a primed cache for exactly that working set.

8.4.1 DEC 8400: Local memory system performance

Figure 8.1 shows the measured load bandwidth with a single processor of an otherwise unloaded DEC 8400. This figure depicts a comprehensive picture of the DEC 8400 memory hierarchy with bandwidth data for all levels of the memory hierarchy. The horizontal plateaus at 700 MByte/s, 120 MByte/s and 28 MByte/s, show the level of memory system performance for different sizes of working sets. The grey bands in the graph indicate the largest working set in the graph with the characteristic performance

[3]Because super-scalar processors perform run-time instruction scheduling, it is important that the operand of a load is actually used. The loop is sufficiently unrolled to hide the latencies of the loads and floating point operations. This transformation may affect some L1 cache numbers, because of issue bottlenecks, but allows us to report true, achievable bandwidth numbers for all other parts of the memory hierarchy.

of (i) L2 cache accesses, (ii) L3 cache accesses, or (iii) DRAM memory accesses. A slope of increasing performance marks the end of the access pattern axis toward lower strides. Its steepness indicates improved bandwidth for loads with contiguous accesses and accesses with small strides. A selection of even, odd, and prime strides permits us to detect performance gains and losses due to a banked memory system. The flat performance for large working sets shows that the interleaved DRAM memory is handled internally and is not visible — except for the cases along the border of the characteristic working sets for different cache levels.

Maximum memory performance for loads is approximately 1100 MByte/s in small working sets that fit entirely into the L1 cache. There is no penalty for higher strides, but the bandwidth becomes difficult to measure due to loop overhead and other constant overheads in the micro-benchmark. An application may experience a bandwidth of 1100 MByte/s out of L1 cache and 750 MByte/s out of L2 cache for large strides even if these bandwidths cannot be measured with a micro-benchmark. For loads out of L3 cache, I experience the peak of 600 MByte/s for contiguous accesses only, while strided accesses fall down to 120 MByte/s. This behavior is caused by the large cache lines at that level and by the read-ahead unit of the L2 cache, which consumes load bandwidth unnecessarily to read-allocate the whole cache line, although only a single word is used.

The measurements in Figure 8.1 show the memory system performance of a single processor while other processors are idle. I also ran the same micro-benchmark with all four processors accessing local caches and DRAM memory independently at the same time. The performance results had a much larger variability but behaved as expected. Because the DEC 8400 has a shared memory system, a decrease in bandwidth for accesses to the shared DRAM memory is expected; the local caches continue working at full speed. The bandwidth for the L1, L2 and L3 cache stays almost the same, while the bandwidth for strided accesses to the DRAM memory decreases by about 8% for contiguous accesses and 25% for strided accesses under full load on all four processors.

The bandwidth for large transfers to and from DRAM memory gives only a partial picture. As seen in Figure 8.1, the machine has a distinct four-level memory hierarchy, which results in big performance advantages for accesses with temporal locality of reference. I therefore consider the four characteristic working sets for a simplified characterization of this fairly complex memory system. The curves in Figure 8.2 are four interesting cuts through the transfer performance surface along the access pattern axes. Each line covers the performance of one of the four characteristic working sets, chosen according to the size of the caches. A working set of 8K means load and the store accesses of a copy hit the L1 cache, a set of 64K guarantees hits of the L2 cache, 4M hits the L3-cache and 65M causes all accesses to go to DRAM memory.

The detailed load performance numbers from L1 and L2 cache are only partly accurate in these graphs. It remains a programming challenge to profile the whole memory hierarchy accurately with a single program, and even a well-tuned program falls sort of the requirements for L1 accesses with large strides and small working sets. The startup overheads of the loops and the fixed overheads of the timers are simply too big to produce any meaningful performance numbers for working sets of 512 bytes with larger strides. If I repeat the experiment millions of times, the total execution time becomes longer, and interfering activities of the operating system disturb my measurements. Given the design of the L1 cache, in theory the top of the performance surface in Figure 8.1 should not look like a ridge but like a flat surface that depicts exactly the level of peak bandwidth for all strides and working sets smaller than L1 cache. The different shape of the performance graphs is due to deficiencies in my micro-benchmark or due to the simple fact that the peak speed just can not be measured directly on real hardware.

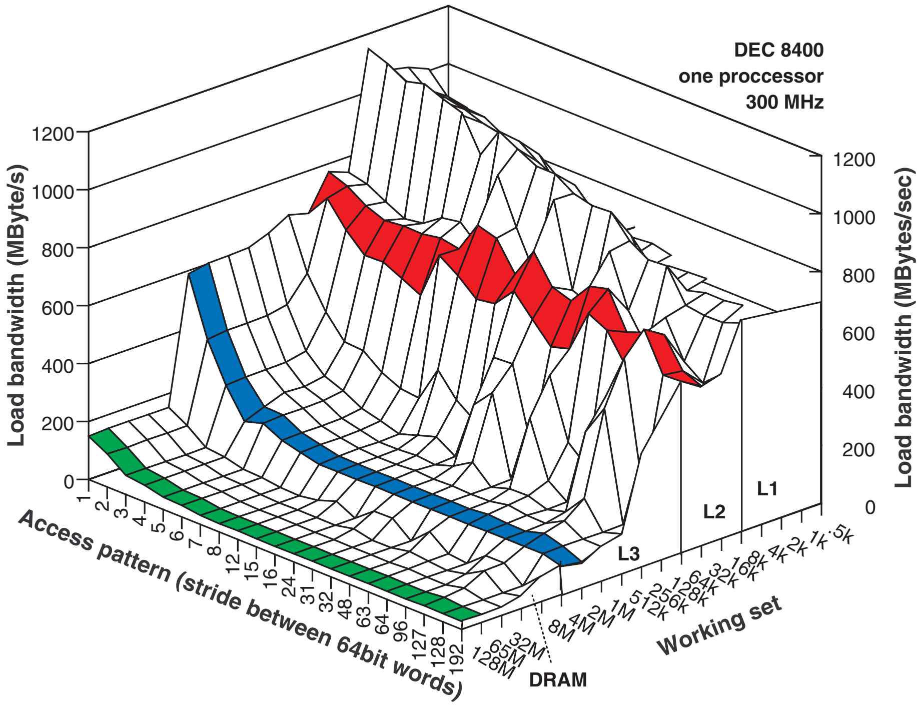


Figure 8.1: Bandwidth of loads for different access patterns (strides) and different working sets on the DEC 8400. One processor runs memory benchmark, other three processors are idle.

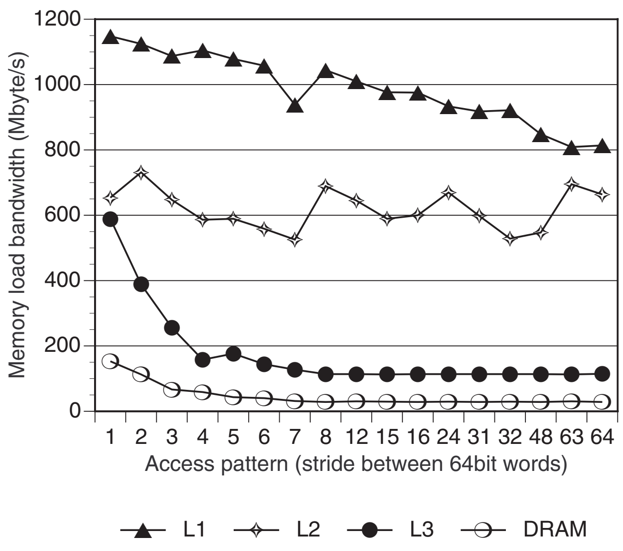


Figure 8.2: Measured access performance for strided loads on the DEC 8400 at four characteristic working sets.

Extending local loads to a local copy

I extend the bandwidth characterization from the simple load ${}_xC_0$ in Figure 8.2 to a full load-store copy transfer with strided loads and contiguous stores in Figure 8.3. Because both streams are now working on the caches and evicting the working set of the other stream, the characteristic working sets are reduced to 2K, 32K, 2M — however, the DRAM working set remains the same. For contiguous accesses there is still a significant difference between the direct mapped, small L1 cache and the larger three-way associative L2-cache, both implemented within the 21164 microprocessor chip. For strided accesses this difference disappears, since the store stream ends up in L2 (L1 is write through). The reads from off-chip, L3-cache and from DRAM memory are significantly slower unless some sort of read-ahead logic detects contiguous accesses (stride 1 or 2). On the store-side of the performance picture, there is little benefit of the L1-cache, since it is write-through to L2-cache. The write performance to off-chip L3-caches and DRAM memory is quite poor even for contiguous writes, therefore copies can never be as fast as simple loads.

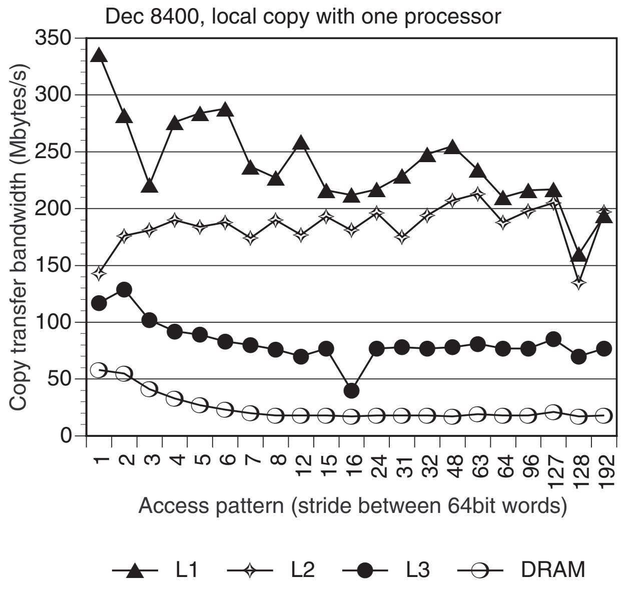


Figure 8.3: Measured local copy performance for strided loads on the DEC 8400 at four characteristic working sets.

8.4.2 DEC 8400: Remote accesses — memory system performance

To measure communication performance, these memory micro-benchmarks include remote memory accesses. On a symmetric shared memory multiprocessor, this mode of operation is achieved when one processor is producing data by storing it while another processor retrieves the same data elements. To ensure race-free behavior, reading takes place after the two processors reach a synchronization point. I measure the transfer bandwidth of the second processor while it is pulling the data over.

Figure 8.4 shows the remote memory performance for the DEC 8400. I see a multi-tiered performance picture depending on whether the working set can be held in fast SRAM cache or slower DRAM main memory. However, notice the entirely different scale for the access bandwidth! The maximal performance for remote memory accesses is down to 140 MByte/s from 1100 MByte/s for local accesses. For accesses out of DRAM, performance is about 22 MByte/s. These results are not surprising since remote accesses not only cross the chip boundaries of processors but also travel across the bus of the shared system and involve coherency protocols. The coherency mechanism detects misses on shared data and pulls the necessary cache lines over from a DRAM memory bank or from the caches of a remote processor board. The DEC

8400 does not have support for pushing data into memory or caches of a remote processor.

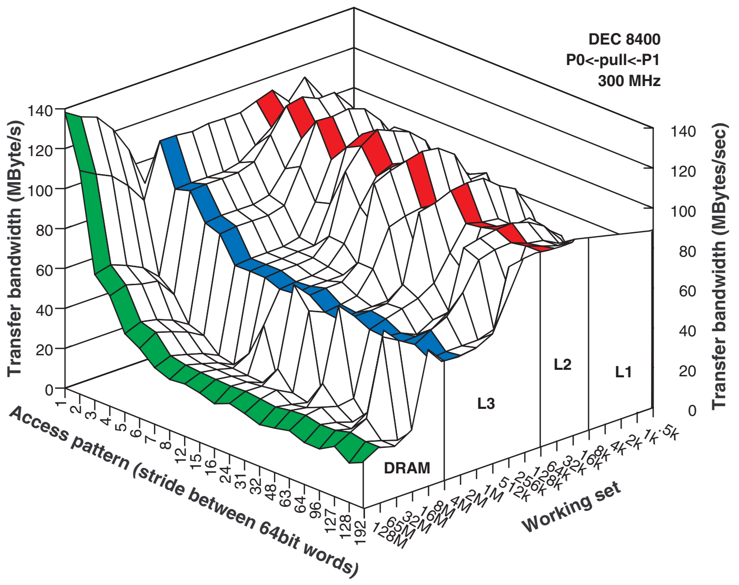


Figure 8.4: DEC 8400: Remote load bandwidth for different access patterns (strides) and different working sets. Throughput for transfers from processor P1 to processor P0.

Similar to the local memory-to-memory copy performance, the remote copy transfer performance of the DEC 8400 depends on the characteristics of a memory hierarchy and delivers different performance for different working sets. I measured and graphed the performance according to the copy transfer model in Figure 8.4. Please note that the scale is different from local copy transfers.

Again, a more detailed graph in Figure 8.5 is limited to the most interesting cuts through the surface along the characteristic working sets. For contiguous accesses, all transfers reach between 90 and 140 MByte/s per node. Transfers from DRAM are actually faster than transfers from remote caches. For larger strided transfers, there is an advantage of communicating remote cache to local cache rather than DRAM memory to local cache. The alternative of keeping all data in local and remote caches has been explored in the ALLCACHE architecture of the KSR [43]. The L1 and L2 caches are likely to be too small for holding even small arrays of distributed data. The L3 might be large enough in some cases. For larger strides the performance improvement for cache-to-cache transfers is a factor of three over the transfer through main memory.

8.4.3 T3D: Local memory system performance

The T3D exhibits a much simpler picture; the performance of local memory accesses is shown in Figure 8.6. With distributed memories, the per-node performance of the local memory accesses looks exactly the same, whether just one or all 512 processors of an entire machine execute programs. The simpler node architecture with a two-tiered memory hierarchy exposes a sharp performance drop when I exceed the working set of the L1 cache and hit upon DRAM memory. An important characteristic of the Cray T3D node design is an external read-ahead logic that contributes to the steep slope of higher performance for contiguous DRAM accesses. Figure 8.6 reflects this design optimization: contiguous loads from local DRAM memory on the Cray T3D are about 30% faster than in the DEC 8400 — despite the T3D's older design and slower clock rate.

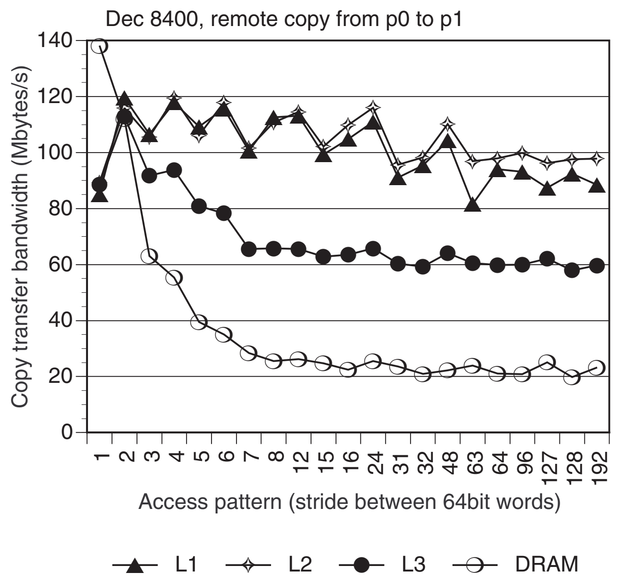


Figure 8.5: Measured performance of remote copy transfers between two processors (P0 and P1) of the DEC 8400 for different strides and for different characteristic working sets.

8.4.4 T3D: Remote accesses - communication system performance

The memory on the T3D is distributed among the node processors. Despite the logical model of a global address space, performance considerations force the T3D compiler to move the data from remote memory to local memory before computing on it. The explicit transfers from local to remote memory in a distributed memory machine have the advantage that there is a choice between fetching data elements from remote memory versus depositing them to remote memory.

I found that for compiler-generated code, deposits based on remote stores are preferable for performance reasons. Naive, compiler-generated remote loads are possible, but they result in communication performance that is an order of magnitude below the network bandwidth — unless the pre-fetch pipeline is used properly. I do not know of a node compiler that produces code for these loads, and programming all compiler communication primitives in assembly language seems too hard to be worthwhile. I concentrate on deposits in Figure 8.8, but for completeness I show some fetch data based on Cray's hand-coded shmem_iget primitives in Figure 8.7. (The Z axis of these two figures is scaled to allow easy comparison with Figure 8.4.)

In Section 8.5.2 I describe all copy transfers based on deposits to characterize communication performance of the Cray T3D.

8.4.5 T3E: Local memory access performance

Figure 8.9 depicts the memory bandwidth for the T3E. The simple node architecture is reflected in the graph: there are three performance levels, exposing a sharp drop from cache access to DRAM memory access when I exceed the working set of the L1/L2 caches. Not surprisingly, the local memory access performance of the T3E resembles the picture of the DEC 8400 in the performance of its L1 and L2 caches. Any differences for small working sets are due to idiosyncrasies in the measurement environment, i.e. the lack of inlined timer functions, and the (still) larger loop overhead inflicted on programs by the Cray

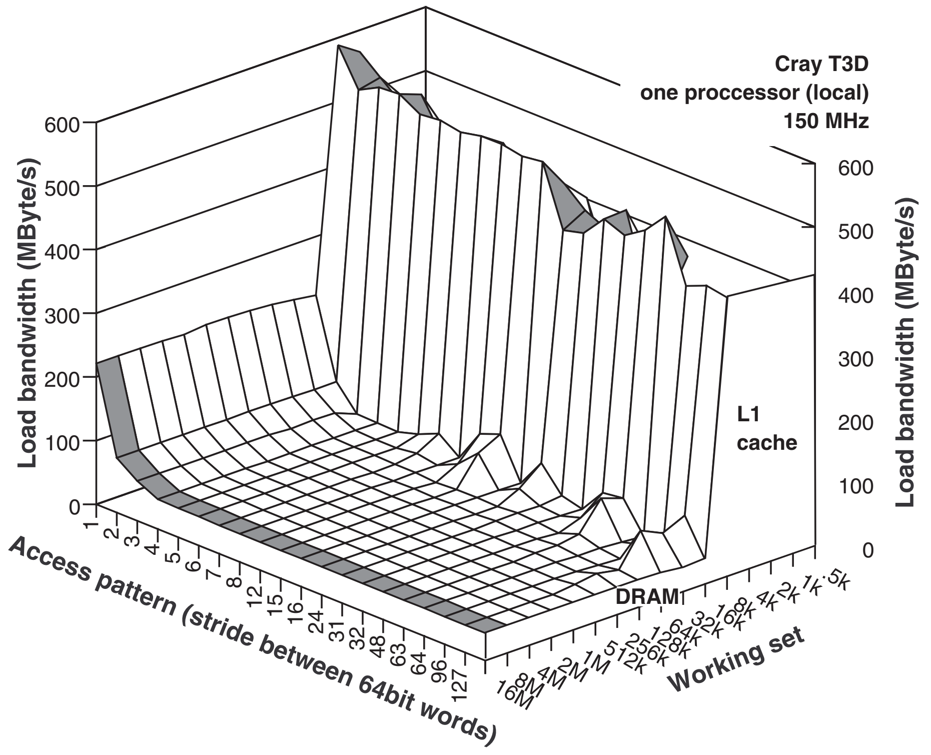


Figure 8.6: Cray T3D: Load bandwidth for different access patterns (strides) and different working sets. One processor runs memory benchmark, and the other three processors are idle.

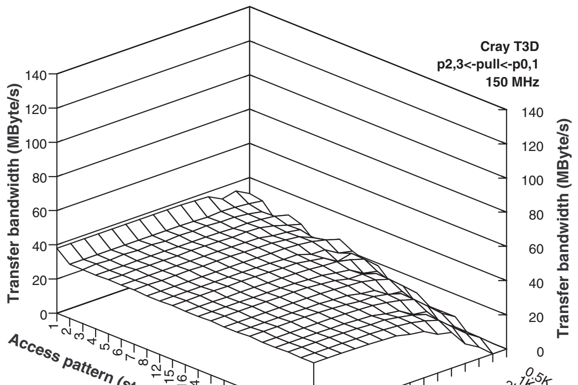


Figure 8.7: Cray T3D under fetch model: Transfer bandwidth for different access patterns (strides) and different working sets obtained through remote loads.

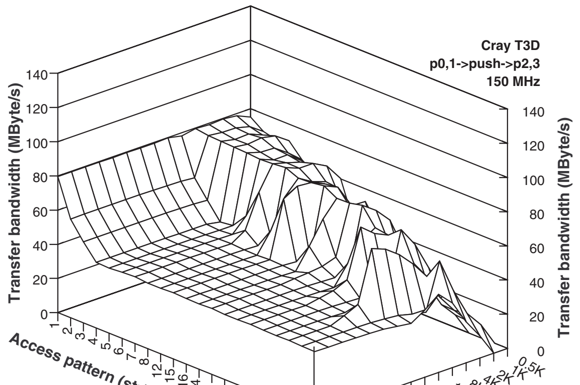


Figure 8.8: Cray T3D under deposit model: Transfer bandwidth for different access patterns (strides) and different working sets, obtained through remote stores.

compiler. More careful benchmarking on the final machine will eliminate the apparent lack of in-cache performance for small working sets.

The important differences in the DEC 8400 are (i) the lack of L3 caches (ii) and the better support for contiguous access streams from main memory. While the DEC 8400 achieves just 150 MByte/s for contiguous loads out of DRAM main memory, the T3E node is capable of load transfers of up to 430 MByte/s. In other words, the memory system bandwidth of DRAM on the T3E compares well with the memory system performance out of the local L3 cache on the DEC 8400 (which is about 600 MByte/s). Again this performance is due to the support hardware for streaming on the T3E (i.e. the pass 3 hardware I ran my tests on; I also measured about 120 MByte/sec on an earlier test-vehicle with streaming support disabled).

While I observe close to a factor of two improvement of L1/L2 cache performance and streamed access to DRAM between the two generations of Cray machines (in line with the doubling of the clock rate), I see no improvement of the strided accesses out of DRAM. These accesses seem stuck at about 42 MByte/s on the T3E (43 MByte/s on the T3D). Although this performance still compares favorably to the DEC 8400 performance of 28 MByte/s, it may nevertheless spell disappointment for some applications that move from the T3D to the T3E.

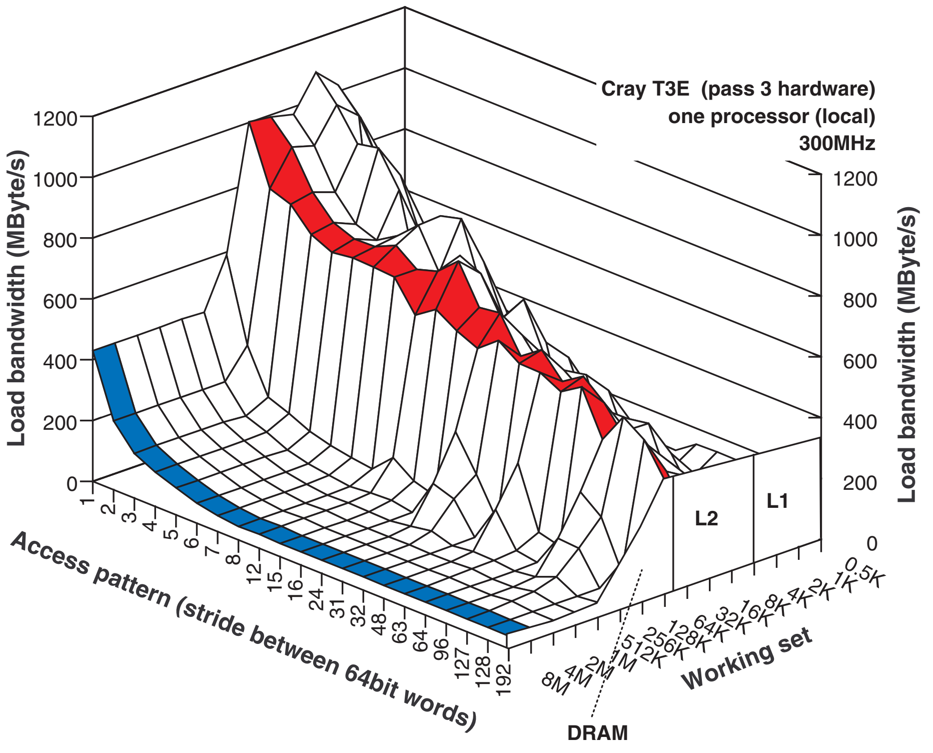


Figure 8.9: Cray T3E: Load bandwidth for different access patterns (strides) and different working sets. One processor runs the memory benchmark, and the other three processors idle.

The T3E memory system performance numbers given here are still inaccurate for larger strides and small working sets. I were not able to inline the timing calls or hand-optimize the loop overhead. Once the final machine and the final compilation tools are installed and available to the general public, a better graph can be drawn.

8.4.6 T3E: Remote memory system performance

The T3E represents a further step from a distributed memory machine to a coherent shared memory multiprocessor. Its global address space can now map the entire memory in the machine, but a compiler is still forced to copy the data from remote memory to local memory (or at least the 512 E-registers) before computing on it. The characterizations reported in this section are based on the shmem_get and shmem_put routines provided by Cray which I treated as black box building blocks.

Unlike on the T3D, the deposit model (i.e., using remote stores) enjoys no performance advantages over the fetch model (i.e., using remote loads). Figures 8.10 and 8.11 illustrate this point. Both modes of operation perform impressively at 350 MByte/sec for contiguous data transfers (stride 1). This is more than four times the bandwidth in the Cray T3D and twice the bandwidth in the DEC 8400. Part of the impressive improvement over the T3D is that in a T3E, every processor has a network node, while in the T3D two processors share a network node. However, the gap between contiguous streams and strided accesses has widened to over a factor of two. The ripples in Figure 8.11 indicate that the memory system at the destination node has difficulties storing data at full network speed if the same bank is hit in consecutive receives. This observation indicates that fetches are even more advantageous for strides than deposits. Therefore the back-end of the Fx compiler will generate fetch code for the T3E while sticking with deposit code for the T3D.

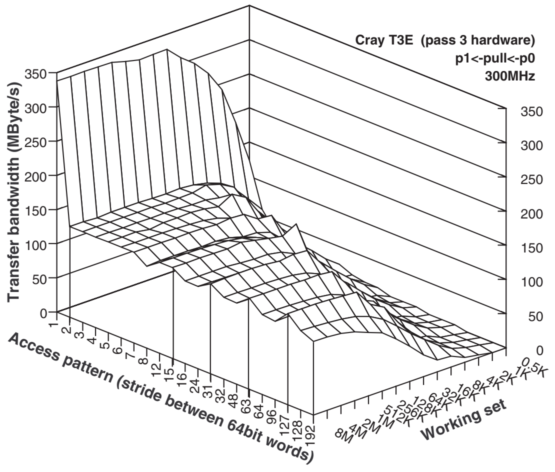


Figure 8.10: Cray T3E under fetch model: Transfer bandwidth for different access patterns (strides) and different working sets obtained through remote loads.

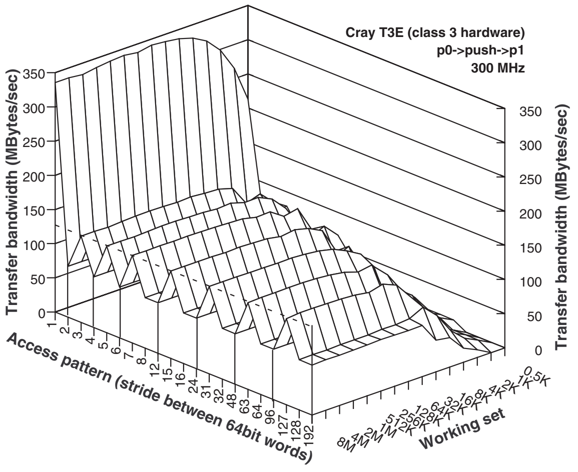


Figure 8.11: Cray T3E under deposit model: Transfer bandwidth for different access patterns (strides) and different working sets, obtained through remote stores.

8.4.7 Paragon: Local memory access performance

The performance characteristics of the local memory system in the Intel Paragon are similar to the one in the T3D. See Figure 8.12 and compare it to Figure 8.6. Accesses to the L1 caches are fast and nearly independent of the access pattern (stride). Once the cacheable working set is exceeded, the performance falls down to DRAM speeds. Given that the i860 processors in the Paragon are running leisurely at 40 MHz, the difference between L1 and DRAM speed is less pronounced than for the DEC Alpha-based systems with processors racing at 150 MHz (T3D) or even at 300 MHz (T3E). Caches help with read-ahead for suitable load streams and accelerate contiguous loads by a factor of 2.5 over the strided case —

from 20-30 MByte/s to 60 MByte/s. With increasing strides, the TLB misses become more common and their overhead slows down the memory accesses. This is not the case if memory is accessed with physical addresses (e.g., in the SUNMOS kernel) or if the operating system configures the TLB for segmented translation. Unfortunately neither mode of operation is used in the Paragon production systems. As for the DEC Alpha-based machines, the performance numbers in Figure 8.12 are from compiler generated code.

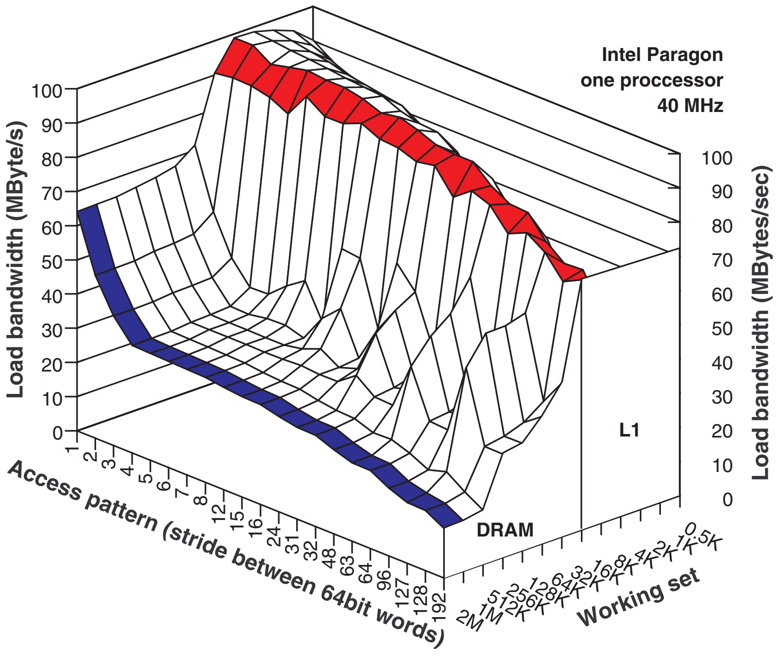


Figure 8.12: Intel Paragon: Load bandwidth for different access patterns (strides) and different working sets. Benchmark executes on a single node.

The Intel Paragon compilers are not able to select and schedule any pipelined instructions for the i860 instruction set. Therefore no compiler generated benchmark can reflect the maximal performance based on pipelined loads. Performance numbers obtained from an assembly benchmark complement the figures measured by the compiler benchmark. Pipelined loads are not cacheable. Their performance is nearly independent of the working set. Figure 8.13 presents the measured performance curves for pipelined loads and regular cacheable loads for a comparison. For regular loads, a small and a large working set are investigated.

A parallel system with a similar processor node architecture is MANNA [44]. Its building block is a board with twin i860 processors and a network interface in an internal bus.

8.4.8 Paragon: Remote memory system performance

For the modeling and benchmarking of remote transfers, I assume that two nodes agree on a large data transfer. The sending node sends one long message containing the whole data block to the receiver. Both the sender and receiver rely on a sufficiently flexible mechanism, rather than on the fastest mechanism, to perform transfers with strided and indexed access patterns. This benchmark involves strided accesses and therefore messages must be injected and extracted by the processor rather than by the faster, but more limited line transfer units (DMAs). Depending on the end at which the strides are applied — either the sending end or the receiving end — the processor at that end of the transfer becomes the bottleneck. The

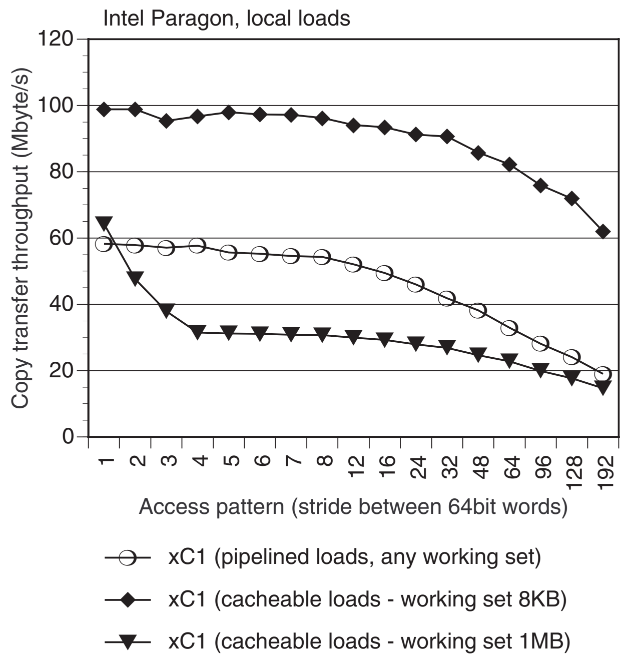


Figure 8.13: Intel Paragon: Load bandwidth for pipelined loads and cacheable loads for different access patterns (strides). The working set is large for pipelined loads, small and large cacheable loads.

Paragon interconnect fabric itself is much faster and can transfer raw data about four times as fast. The performance curves in Figure 8.14 must be annotated here with a critical remark. Modern shared memory machines, including the DEC SMPs and two Crays under consideration, contain superior hardware support, in contrast to the Paragon, for handling incoming remote memory accesses without processor intervention. Their memory systems contain extra logic to handle such external accesses at minimal overhead. On the Paragon the microprocessor is used to deposit incoming data. The numbers given are for an otherwise idle sender and receiver. In theory, the twin processor architecture of an Intel Paragon has an extra processor to handle background receives, but in practice performance for simultaneous send and receive operations would be significantly lower due to excessive arbitration overheads in the intra node memory bus.

8.4.9 iWarp: Local memory access performance

The performance characteristics of the iWarp memory system hint at a memory system architecture that is almost too good (and too expensive) for the microprocessor it is attached to. The performance of the SRAM-based, “all cache” memory system is almost uniform. Even worse, startup overheads cause a reversed performance characteristic, where performance is better for large working sets than for small ones. In Figure 8.15, the layout of the category axis (working-set) must be reversed for a proper visualization of the data. Performance is the best for large working sets, since there is no caching involved on iWarp. Performance degrades for smaller working sets (towards the front of the graph) because of pipeline overheads in the memory access loops (use of mempipe in the LIW instruction set).

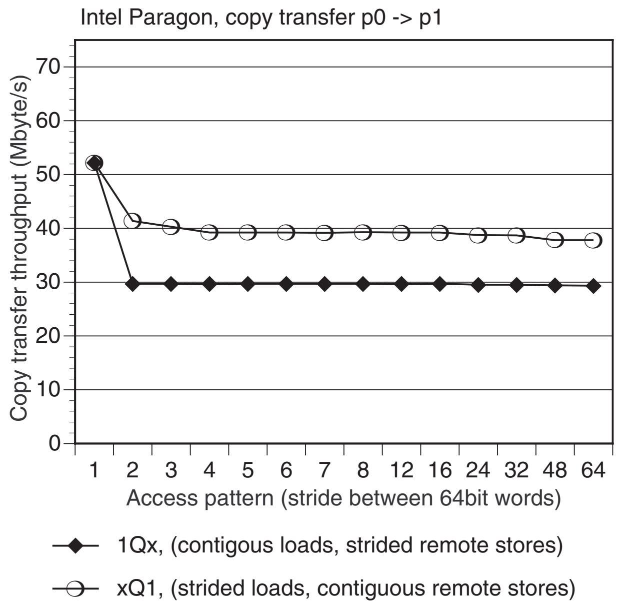


Figure 8.14: Measured performance of remote copy transfers between two processors (P0 and P1) of an Intel Paragon for different strides. Two options for strided accesses are considered, strides at the load end (${}_xQ_1$) and strides at the store end (${}_1Q_x$).

8.4.10 iWarp: Remote memory system performance

On the iWarp the remote accesses are measured under conditions similar to those for the Intel Paragon. A single long message is transferred between two nodes. The support for connections (pathways) makes this particularly simple, so the software can just open a connection and perform the whole copy transfer. This mode of operation fits in extremely well with the copy transfer model, and in practice the overhead to open such a connection can be discounted since it is amortized over an entire large message. Processor-driven sends and receives are used for this test, again with a fully dedicated processor on both the sender and the receiver side. Spoolers (DMAs) would work on contiguous transfers, but there would be no advantage in such a setup. Processor-driven communication can saturate a link at 40 MB/s with ease (at least after the compiler writer knows about all the implementation glitches of the iWarp component). Depending on the access pattern, the sending end or the receiving end uses strided accesses in the copy loop for injecting or extracting the data to the network. The post-incremented loads and stores of the iWarp's extended RISC instruction set incorporate address computation at no extra cost. With no overheads left, the copy transfers indeed occur at full link speed, resulting in the rather boring, flat performance curve in Figure 8.16. The combination of the high-powered SRAM memory system, gate registers for communication, and a unique high-speed interconnect network results in a most impressive remote memory access performance. The full performance is sustained even under complex access patterns involving strides and indices.

8.5 Comparison of memory performance among the DEC Alpha systems

Three out the five systems in this characterization use a DEC Alpha processor, although the T3D uses an older implementation. The similarity in the processor architecture and the continuity in the line of these DEC Alpha systems make it particularly interesting to study and compare the performance of their

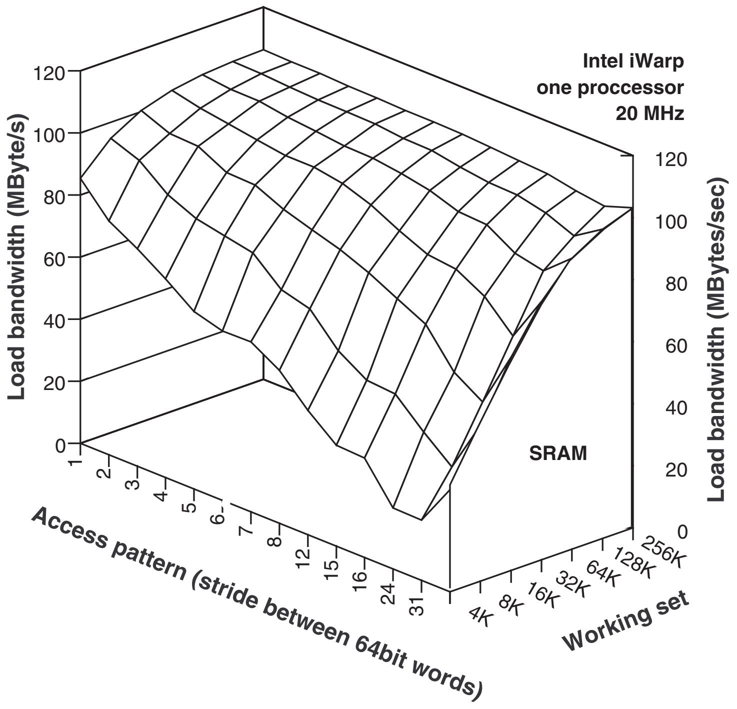


Figure 8.15: Intel iWarp: Load bandwidth for different access patterns (strides) and different working sets. Benchmark executes on a single node. Important note: The axis of the working set had to be reversed to allow proper visualization.

radically different memory system designs.

Scientific application codes for parallel computers handle large arrays of distributed data. I encounter a few characteristic communication patterns when arrays are redistributed. For many distributions, every processor must exchange data with every other processor. These "all-to-all personalized communication" (AAPC) operations have received considerable attention by researchers. Transposes are one example of an AAPC; since the actual execution of these operations does not depend on the data values, a transpose is a good generic test case to assess the joint strengths and weaknesses of the communication and memory system. The performance of array transposes is largely determined by the ability of the DRAM memory system to handle local and remote copy transfers with strides (dense matrices) or indices (sparse matrices). Therefore, this characterization of the memory system goes beyond pure load and store bandwidth to measure the copy bandwidth. Since end-to-end transfers in compiled parallel programs often involve strides, we graph access patterns for various strides, from contiguous up to constant strides of 64.

In many computation and communication steps, the amount of data (re)distributed, called the communication working set, is much larger than the total amount of cache memory in the machine. With a largest cache size of 8 KByte on the T3D and 4 MByte on the DEC 8400, a working set of 65 MByte per processor is sufficient to force every copy operation to go from DRAM memory to DRAM memory. In this section we select a few key working sets from the 3D-surface plots of my general memory characterizations in Figures 8.1 to 8.15 and discuss the memory system performance in more detail. The characterizations are according to the basic copy transfer model [98] (no working set parameter, full copy operation — read and write) and focus on large copy transfers with no reuse of data and without temporal locality in the caches.

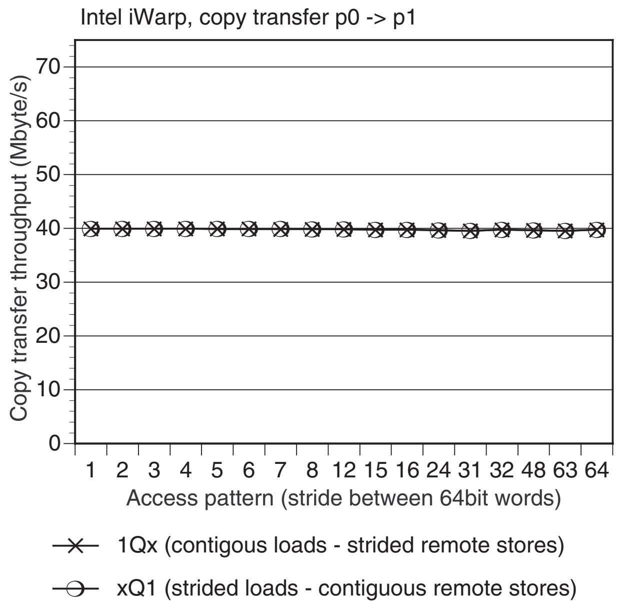
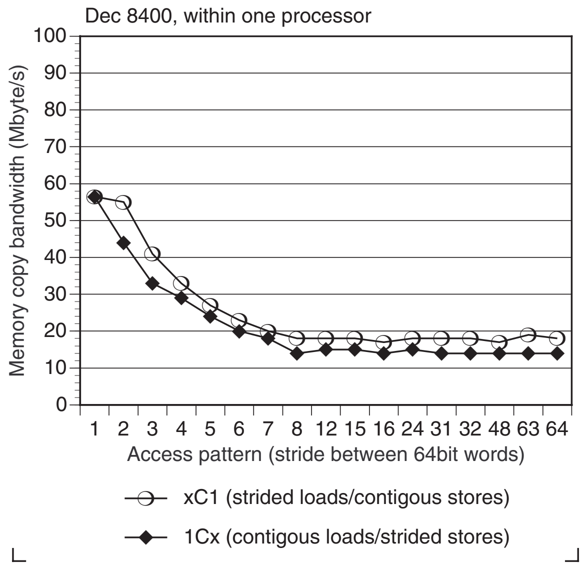


Figure 8.16: Measured performance of remote copy transfers between two processors (P0 and P1) of an Intel iWarp for different strides. Two options are considered for the access pattern, strides at the load end and strides at the store end.

8.5.1 Copy transfers in local memory

Figures 8.17 and 8.18 depict the measured throughput of a local memory copy with either strided loads or strided stores. On the DEC 8400 and the T3E, there is little difference in performance between local copies using strided loads versus local copies using strided stores. The caches help equally with read-ahead and write-behind to accelerate transfers with low strides. A DEC 8400 can copy contiguous blocks at about 57 MByte/s and strided data at about 18 MByte/s. Given the high clock rate and the high nominal performance of the bus system of the DEC 8400, these memory bandwidth numbers are rather disappointing. On the T3D I see a significant improvement in bandwidth for contiguous loads due to the read-ahead logic. The T3D node architecture is able to copy contiguous memory blocks at 100 MByte/s despite its lower clock rate of 150MHz. Furthermore, well-pipelined writes through a write-back queue allow strided stores at up to 70 MByte/s, which is almost three times the speed of the DEC 8400. The T3E has an impressive copy bandwidth of 200 MB/s for contiguous blocks, but unfortunately the picture for strided access resembles more the DEC 8400 than the T3D. The write-back caches prohibit efficient strided stores. This observation holds for compiler generated copy loops and may not be accurate for assembler generated programs that manage to work around the default write-back cache policy. The good performance of some communication libraries suggests that assembly loops with adequate prefetching can further activate the streaming support and perform much better than a C program compiled with the vendor's node compiler.

For the DEC 8400 the bandwidth results for large transfers to and from DRAM memory (basic copy transfer model) show only part of the picture. This machine has a distinct four-level memory hierarchy; if a transpose operation can keep its working set in a given level of the memory hierarchy, big performance advantages may be obtained. The L1 and L2 caches are too small to make this option attractive, but blocked algorithms for the L3 caches could yield interesting performance numbers. The characterization in the extended model (Figure 8.1 in Section 8.4.1) contains the data to assess the performance gain of



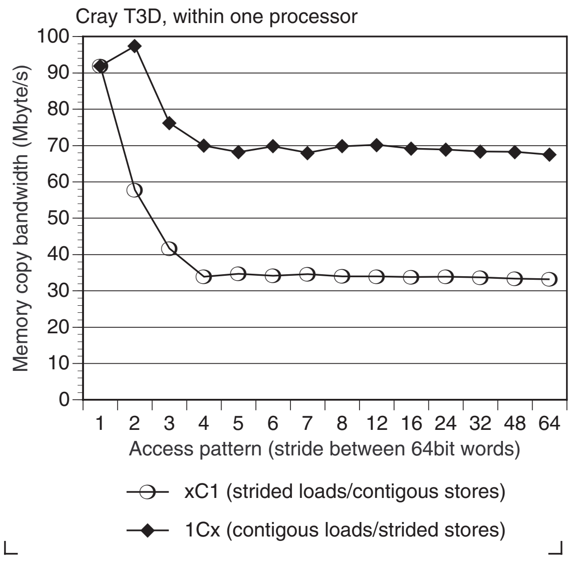


Figure 8.17: Measured performance of the local memory system for large transfers, with either strided loads or strided stores for the DEC 8400 (left) and the Cray T3E (right).

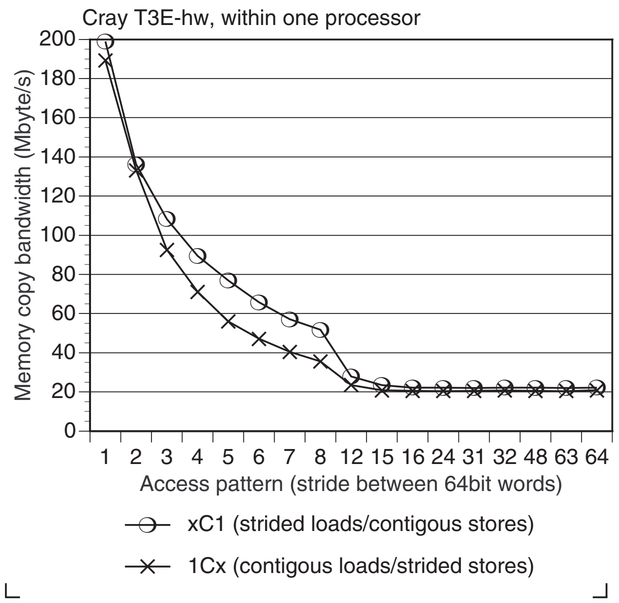
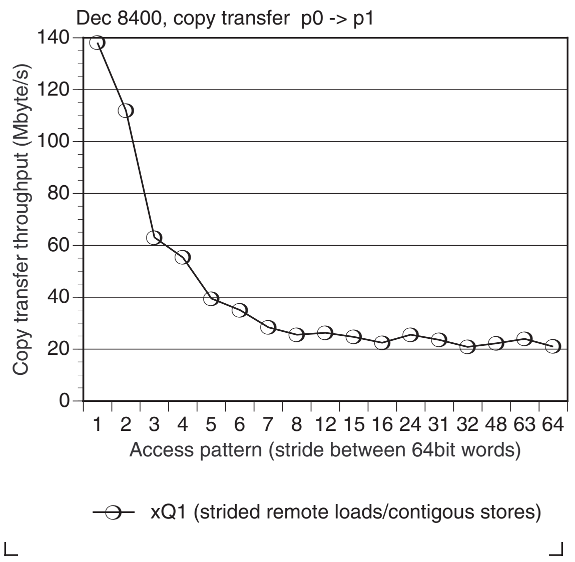


Figure 8.18: Measured performance of the local memory system for large transfers, with either strided loads or strided stores for the Cray T3E. Note the different scale.

such blocked algorithms (just follow the marked cut for the L3 working set (4MByte) in the figure through the different strides).

8.5.2 Copy transfers to/from remote memory

On the Cray T3D and the Cray T3E, all remote store/load operations are explicitly programmed, while on the DEC 8400 the cache coherency protocols decide to pull/push data between processors for certain load and store operations. In this study I focus solely on throughput for moving data from A to B; therefore I can easily compare the two architectures based on remote copy transfers. On the Cray T3D and the Cray T3E, I have the option of performing either strided loads or strided stores to implement a particular transpose operation. For a characterization most relevant to scientific applications with large distributed arrays, it is sufficient to compare transfers with big working sets. In Figures 8.19 and 8.20 I graph transfer performance for a 65 MB working set on the DEC 8400, the Cray T3D and the T3E. The DEC 8400 and the Cray T3D handle contiguous data at about the same speed. For strided data, however, the T3D has a clear advantage as higher strides get involved. If copy transfers of transposes are properly optimized using strided stores on the T3D, they can be performed at about 55 MByte/s, while on a DEC 8400 the bandwidth of such transfers is limited to about 20 MByte/s. The performance on the T3E is in a class of its own. It can transfer 350 MByte/s for contiguous blocks and falls down to 140 MByte/s or 70MByte/s for strided accesses (depending on how the transfer is programmed). The ripples for odd strides suggest that the memory system at the destination node needs to avoid bank conflicts to keep up with the network speed. This is not surprising, since in the most recent generations of parallel system interconnects, the network performance has improved faster than memory system performance.



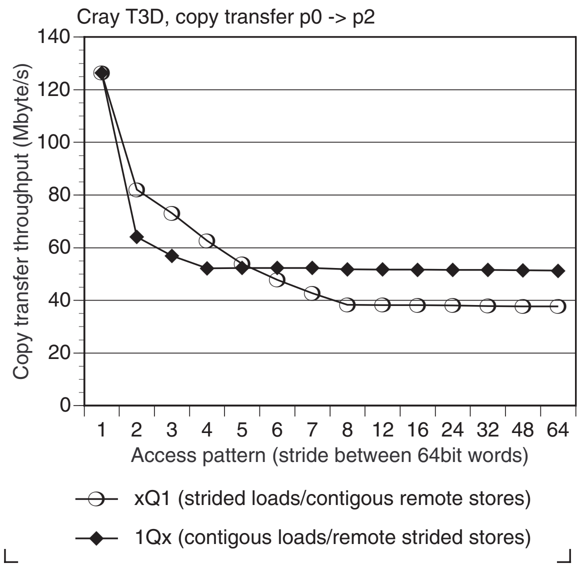


Figure 8.19: Measured performance of large remote copy transfers (i.e. between processors) on the DEC 8400 and the Cray T3D, for different strides.

Similar to the local memory to memory copy performance, the remote copy transfer performance of the DEC 8400 depends on the characteristics of a memory hierarchy and delivers different performance for different working sets entirely within the L3 cache. Again the full characterizations with different working sets in Figure 8.4 reveal that strided remote transfers can be done faster from L3 cache if a global communication operation can be blocked. The characterization quantifies the advantage for this interesting compiler optimization.

The measured performance for a remote copy transfer sets an upper bound on memory system performance in parallel applications, since my simple measurements fail to capture some sharing of certain

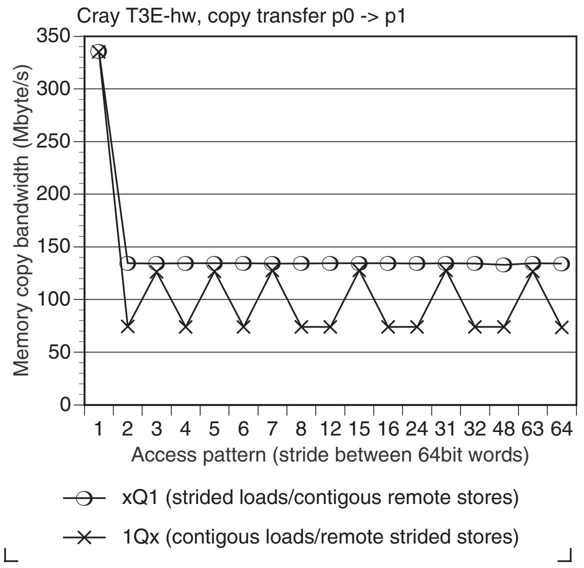


Figure 8.20: Measured performance of large remote copy transfers (i.e. between processors) on the Cray T3E for different strides. Note the different scale.

communication resources on both machines. On the T3D I used only one out of the two processors that are sharing a common network access (and with it a link). On the DEC 8400, I measured just one single processor or a single producer/consumer pair while copy transfers between two processors with no other workload executed on the machine. On the T3E there is no contention, and the remote copy transfer performance is expected to scale up to a 512 processor ($8 \times 8 \times 8$) torus, before bisection limits become visible in transposes (AAPC patterns).

8.6 Summary of the comparison

The DEC 8400, the Cray T3D, and the Cray T3E support remote memory operations as means of interprocessor communication. Despite their differences in the support of coherency and startup overheads for communication, the performance of these machines can be characterized accurately with the copy-transfer model, a simple bandwidth oriented model for memory system performance under different strides. The multi-level cache and memory hierarchies of machines like the DEC 8400 make it necessary, however, to extend the basic copy transfer model with an additional working set parameter to capture the potential gains through operand reuse and blocking.

In this characterization and comparison of the local and remote memory system performance, I observe a few interesting characteristics on three machines. These machines belong to different generations and were introduced over a 3-year period (Fall 1993 (T3D), Spring 1995 (DEC), Fall 1996 (T3E)), and looking at the data sheets, I notice a progression towards faster clock rates, larger caches and faster bus or network interconnect speeds. However, these improvements do not translate into comparable across-the-board improvements in processing power, better memory systems performance, and better communication throughput with each generation. Comparing the T3D and T3E, a 2.5 fold increase in processing power is matched by a 2 fold improvement of local memory copy bandwidth and a 2.75 fold improvement in remote memory copy bandwidth. The DEC 8400, on the other hand, shows a significant improvement in local compute performance over the T3D (due to the next generation microprocessor with doubled clock rate), but its memory system performance for large local and remote copies turns out to be inferior to the older T3D. For large strided remote transfers the DEC 8400 achieves only 22 MByte/s per processor, a factor of 2.5 less than the 55 MByte/s measured in the T3D, or a factor of 6.5 less than 140 MByte/s measured in the T3E. An exception to that are contiguous accesses and small strides where the T3D and DEC 8400 perform alike — but still a factor of 2 below the T3E. I attribute these differences to the memory system design philosophies, i.e. a cache focus on the DEC machine and a streams focus on the Cray machines.

Chapter 9

Evaluation

In Chapter 5 and 6 I argue for improved performance due to decoupled synchronization and highly optimized direct deposit data transfers. For the evaluation and experimental verification of my claims I implemented several deposit messaging systems and ran a few application kernels with different communication patterns between processing nodes and these kernels include different access patterns to the communicated data. The systems used are the iWarp, the T3D, the Paragon, and the DEC8400. In this section I present the experimental part of this thesis and show how experimental work furnished many valuable insights into the nature of the communication overheads of message based communication. The experiments prove the effectiveness of the communication optimizations proposed. One of the important contributions of this thesis is my proposal to rearrange the structure of message passing software for the best possible communication performance for compiled parallel program executing on a given class of parallel machines. Those machines have in common that they are designed for parallel computing and have a high performance communication system. A considerable effort is spent to tune the benchmarks for maximal performance and measure the resulting performance as a fraction of the uppermost limits achievable on that given hardware. This includes the time-consuming task of performing all necessary optimizations in every case and an extended effort to create and maintain a controlled environment for measurements at peak speed. Best performance was given priority over full functionality and portability, limiting the evaluation to a few application codes with small to moderate complexity. I understand that a human programmer a not go through this complicated process for each application, but in the context of a parallel Fortran compilation system, such effort can be justified by the reuse of the compiler in many application.

As a drawback of the aggressive optimization and the requirements for a “peak speed” environment, it becomes extremely difficult to incorporate the optimizations immediately into a large software system, such as as complete parallel compiler. Giving away some of the speed advantages would facilitate the tasks of integration, but without an evaluation that carries a particular parallel machine to its hardware limits, only little could be concluded about the best possible structure of a communication system software and about the costs and the benefits of the architectural support on that particular machine. My benchmarks are as complete as possible when it comes to speed. They match the best performance numbers known for the task and the target machine they are written for.

The experimental data in this chapter is taken from three experimental frameworks: (1) A straight comparison of direct deposit messaging versus postal messaging on the Cray T3D. I use a 2D-FFT and a SOR kernel for this comparison and, in addition, I discuss a partial optimization to an finite element application code. (2) An evaluation of modeled and measured performance for different methods of data transfers on the T3D and the Intel Paragon. (3) A comparison of the direct deposit communication back-end versus a cache coherent shared memory back-end for the Fx compiler. For this experiment, performance on a DEC 8400 is compared to a Cray T3D.

9.1 Decoupling communication - deposit messaging

A first experimental framework is designed to quantify the effect of the synchronization model and the overall software structure on communication performance. A vendor supplied conventional message passing system is compared to the fully optimized deposit messaging system and an intermediate implementation designed to expose the performance component of the direct transfers without the global optimization of synchronization.

9.1.1 The T3D as an ideal host for comparisons

To assess the performance impact of different message passing systems, I measure the execution of two application kernels on the Cray T3D. My choice of the T3D is motivated by three reasons:

- The native implementation of PVM 3.0 on the Cray T3D is a good approximation of the postal style of message passing.
- The high-speed communication system (1 Gigabit/s sustained communication point to point) provides a challenging framework. If the different communication styles make a difference on this system, the impact is even higher on systems with lower performance communication systems, since on those systems, the overall contribution of communication to program execution is higher.
- The Cray T3D includes fast hardware barriers, thereby providing us with a platform to explore all options.

9.1.2 Comparing message passing systems

I measure the computation and communication performance of my kernels running under three different message passing systems.

PVM 3.3 The PVM library provides the conventional *send* and *receive* primitives, which combine synchronization and data transfer services. It is important to note that PVM 3.3 is a specialized library version of PVM for communication completely within the T3D distributed memory parallel computer [60], as opposed to a more general version that is also capable of communicating over a network of heterogeneous systems. This PVM library was written and optimized by the vendor under the assumption of *exclusive processor use* and *full access to the communication system*. Thus, all my programs run in physical memory and communicate directly from user space to user space. The results reported for this implementation of PVM are also indicative of the performance that can be obtained with a native implementation of MPI, the evolving message passing standard.

DMSG: Deposit model message passing My second communication service is based on the idea of service decomposition into control and data transfers. DMSG allows deposits into the address space of any other node. At the destination node, the deposit engine executes the remote store asynchronously, without any participation from the receiving node. The sender keeps track of when remote stores complete. For all further synchronization, the compiler relies on the hardware barrier, which can synchronize all processors within a few microseconds.

RRMSG: request/response message passing With the third library I study the case of machines where control and data transfers are optimized separately, but there is no direct support for synchronization through hardware barriers (e.g., Intel Paragon or SP2). I refer to this model as "RRMSG". This style mirrors closely the operation of NX for long messages on a Paragon running under OSF/1 or an iPSC860. For every data transfer there are three control messages: a *request* by the sender for transfer, a *response* in which the receiver confirms buffer reservation, and a final acknowledgment by the receiver confirming the reception of the data. These extra control messages are necessary to free the user from potentially complicated buffer management decisions. In some cases, buffering and copying can be avoided with the exchange of a few synchronization messages. The RRMSG model incorporates ideas from both worlds: the conventional message passing systems (e.g., PVM) and the highly efficient direct transfers of the deposit model (e.g., DMSG).

All three implementations rely ultimately on the built-in T3D *remote store* commands to implement data transfers. The differences that I report in the next section are due to the different synchronization schemes, emphasizing the importance of adequate synchronization support in parallel systems.

9.1.3 Applications used in the evaluation

To evaluate the impact of different passing implementation targets, I chose two kernels for the measurements in depth and look at a real application for a isolated experiment. For the application kernels I chose one with a dense communication pattern and one with a simple, nearest-neighbor pattern. The communication pattern of the full application code is an communication graph resulting from a well partitioned irregular mesh. The density of its communication graph is somewhere in between next-neighbor and fully connected.

2D-FFT The first kernel is a two-dimensional Fast Fourier Transformation (2D-FFT) on an $N \times N$ array of single precision complex numbers. The columns of the array are distributed by block.[1] During the first computation phase, each node independently performs a one-dimensional FFT operation on each column residing on that node. Next comes a communication phase, in which I transpose the array. This transpose results in a dense all-to-all communication, in which each node sends a distinct data block to every other node. After the transpose, I once again perform a set of 1D-FFT operations on each column. Finally, I transpose the array again, resulting in another all-to-all communication. This example also captures the communication behavior of large one-dimensional FFTs. On parallel systems, these are often broken up in this way into artificial "rows" and "columns" for better performance.

A typical application then proceeds with a series of filtering operations and possibly performs another 2DFFT to transform the data back to the original domain at the end.

SOR My second application kernel is an example of such a filter, the computation of a k-point stencil over a two-dimensional $N \times N$ block-distributed array. This kernel is also used in successive over relaxation (SOR) iterative solvers. The stencil computations have simple communication patterns, exchanging data only with a few neighbors (typically one in each direction). The algorithm usually iterates multiple times until convergence is reached; my kernel is measured with 10 iterations of a 5-point stencil, width 4 in each direction. As in the FFT, I mapped only one dimension of the 2D array onto the nodes regardless of the three dimensional physical structure of the T3D. A better mapping could improve the application kernel further.

[1] I assume a column-major memory layout of arrays, as in Fortran. In C, I would distribute by rows.

FEM The application under consideration is a simple iterative solver for the equations of earthquake ground modeling problem. The problem size is varied from $30 * 10^3$ points/$150 * 10^3$ tetrahedra (resolution 5 seconds) to $378 * 10^3$ points/$2.06 * 10^3$ tetrahedra (2 sec) and $2.4 * 10^6$ points/$14 * 10^6$ tetrahedra (1 sec). The average connectivity of the communication patterns is 16-17 for 64 processors and 19-21 for 128 processors. The average per message sizes varies from 28 words for the 5 sec on 128 processors to 904 words for the 1 sec on 64 processors. The algorithm usually iterates through the simulation interval in fixed time steps. The meshes are partitioned and mapped straight to processor numbers without consideration of the 3D topology of the machine. The simulation problems of that project are large and irregular; methods are found in [6] and a characterization of the meshes is contained in [83].

The rest of this section refers to detailed measurements with the two kernels FFT and SOR, while the next section describes FEM under conventional and decoupled implementations of MPI. Table 9.1 shows for both application kernels the overall performance per node and the aggregate performance for 512 nodes. I also list the data rates to show the impact of communication performance in more detail. The relative fraction of communication and local computation work and the effect of communication style on scalability is studied with common problem sizes of one to sixteen million elements (e.g., 4096×4096 for 2D-FFT and SOR). I determined this problem size by looking at grand challenge applications in earthquake and air-shed modeling.

Two-dimensional FFT Problem Size: 4096×4096 on a 512 node T3D				
Msg Passing System	MFLOPS per node	MFLOPS total	MByte/s per node	MByte/s total
PVM	2.1	1103	0.64	326
DMSG	17.1	8755	20.27	10379
RRMSG	13.6	6938	9.39	4809

Two-dimensional SOR (5pt stencil) Problem Size: 4096×4096 on a 512 node T3D				
Msg Passing System	MFLOPS per node	MFLOPS total	MByte/s per node	MByte/s total
PVM	11.7	5977	21.1	10845
DMSG	15.5	7939	48.4	24822
RRMSG	15.4	7882	47.3	24243

Table 9.1: Results for the application kernels for message passing system comparison.

9.1.4 Performance impact of fast synchronization

The scalability of application kernels depends critically on the message passing style and hardware support for synchronization. The breakdown into computation time and communication time shows how the communication part can become the dominant factor in the overall execution times in my FFT and SOR kernel as graphed in Figures 9.1, 9.2, 9.3 and 9.4. The different overhead of communication relative to computation for PVM and deposit message passing is most visible in the FFT kernel with PVM message passing. While with PVM the communication time is almost constant and dominant on large machine

(64-256 processors) it scales with the computation time in the DMSG case. The picture is similar for the SOR kernel, although it is not pronounced as in FFT.

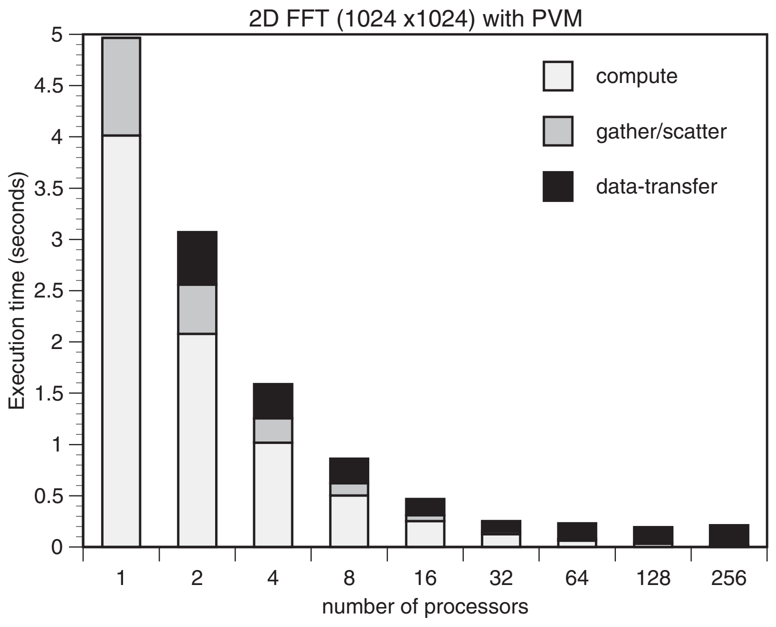


Figure 9.1: PVM: Execution times of 2D-FFT for different Cray T3D machine sizes.

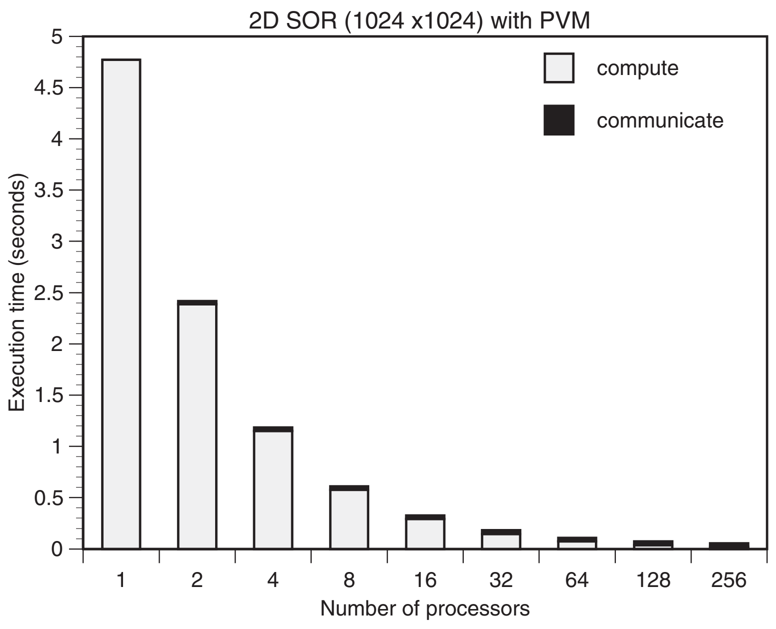


Figure 9.2: SOR: Execution times of 2D-FFT for different Cray T3D machine sizes.

In Figures 9.5, 9.6, 9.7, and 9.8 I graph the total aggregate performance and the per-node performance for the different messages passing systems on different machine sizes. These performance figures include all communication overheads due to parallel execution.

The performance of the 2D-FFT kernel executing on PVM falls sharply as I move to machines with more than 64 nodes, due to lower communication speeds. Better scalability and good sustained per-

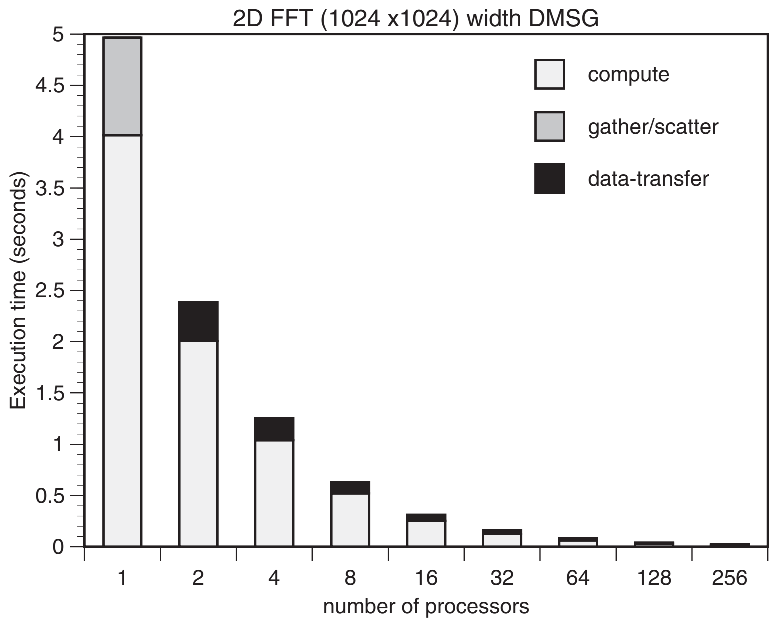


Figure 9.3: DMSG: Execution times of 2D-FFT for different Cray T3D machine sizes.

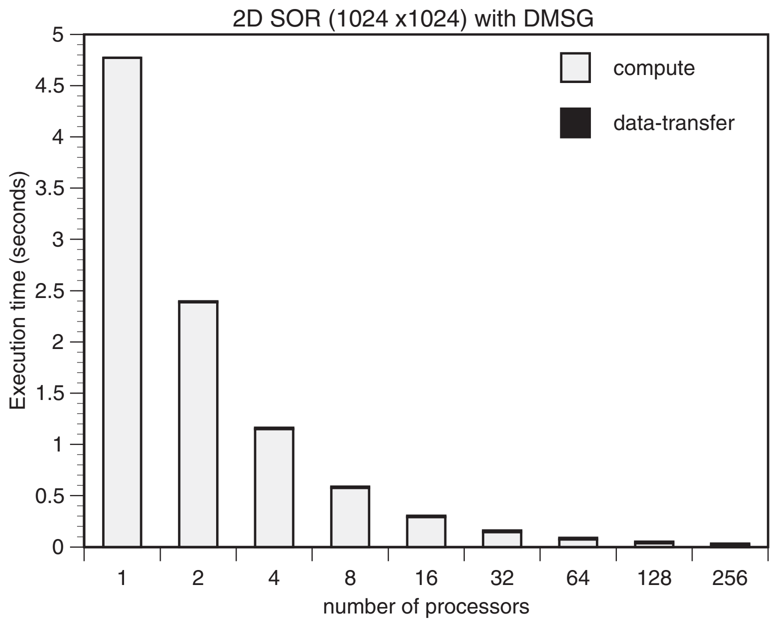


Figure 9.4: SOR: Execution times of 2D-FFT for different Cray T3D machine sizes.

formance can be achieved with the decoupled models that are the basis of DMSG and RRMSG. (See Figures 9.5 and 9.6.)

Due to separate synchronization, DMSG is able to make use of the good architectural support for hardware barriers and fast direct deposit data transfers. In DMSG, the deposit model library, the per-node performance remains near peak for both applications (2D-FFT and SOR), even at machines sizes as large

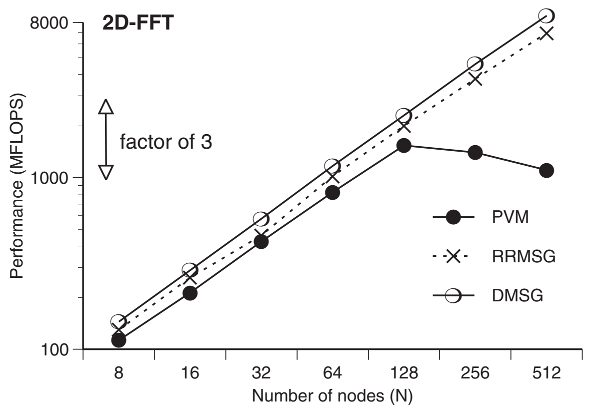


Figure 9.5: Aggregate performance of a 4096×4096 point 2D-FFT for different machine sizes, graphed on a log-log scale.

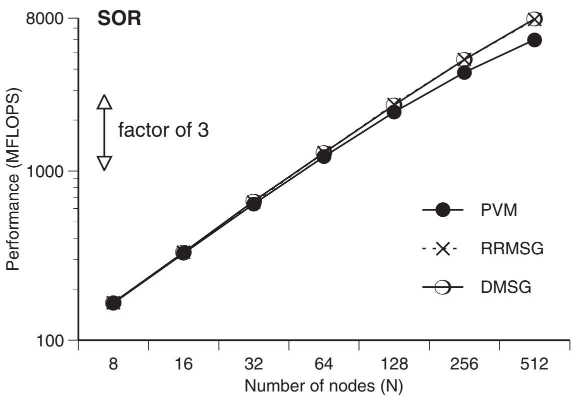


Figure 9.6: Aggregate performance of a 4096×4096 element SOR for different machine sizes, graphed on a log-log scale.

as 512 nodes, as depicted in Figures 9.7 and 9.8.

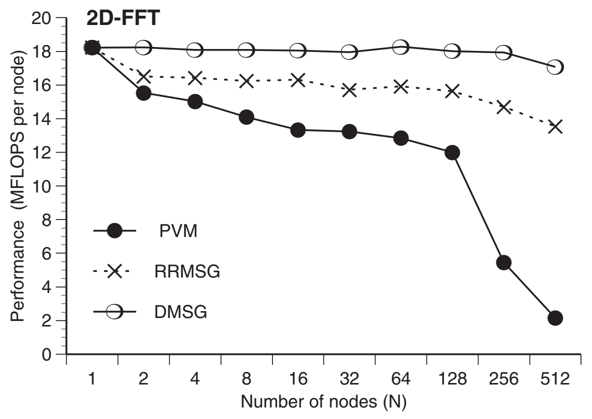


Figure 9.7: Per-node performance of a 4096×4096 point 2D-FFT for different machine sizes.

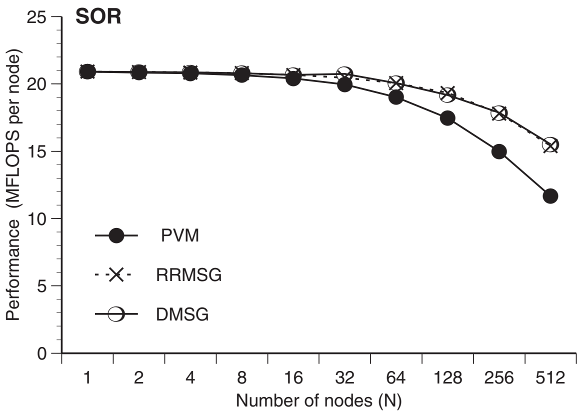


Figure 9.8: Per-node performance of a 4096×4096 element SOR for different machine sizes.

For a more detailed analysis of communication performance, I measure the achievable throughput of both applications during communication phases independent of the computation part. These numbers can be used to relate the net throughput (after all overheads) to the peak communication performance of the T3D hardware, specified at about 130 MByte/s per link, or about 65 MByte/s for each of the paired nodes, if both processors communicate simultaneously.

The communication performance is more important for applications with transpose steps (e.g., FFTs) than for applications with simple next-neighbor overlap exchanges (e.g., solvers like SOR). The performance improvement noted by such applications is mainly due to the substitution of a large number of control messages by hardware barriers in the all-to-all communication step. Furthermore, well-synchronized programs can benefit from the better performance of direct deposit data transfer for strided local access pattern.

The aggregate communication transfer rates are shown in Figures 9.9 and 9.10, and the per-node communication transfer rates are shown in Figures 9.11 and 9.12. Communication performance in PVM seems

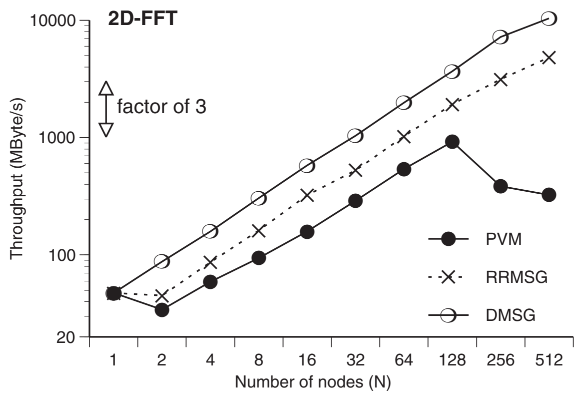


Figure 9.9: Aggregate communication performance of a 4096×4096 point 2D-FFT, graphed on a log-log scale.

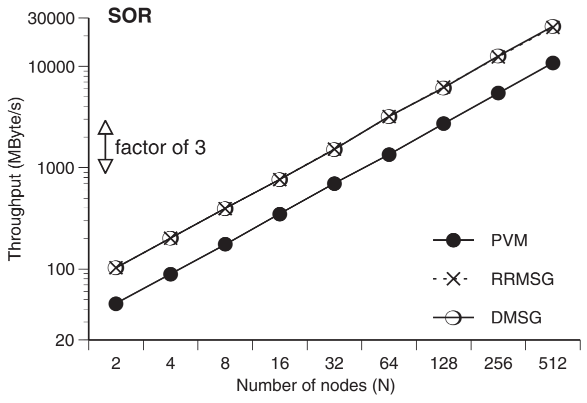


Figure 9.10: Aggregate communication performance of a 4096×4096 element SOR, graphed on a log-log scale.

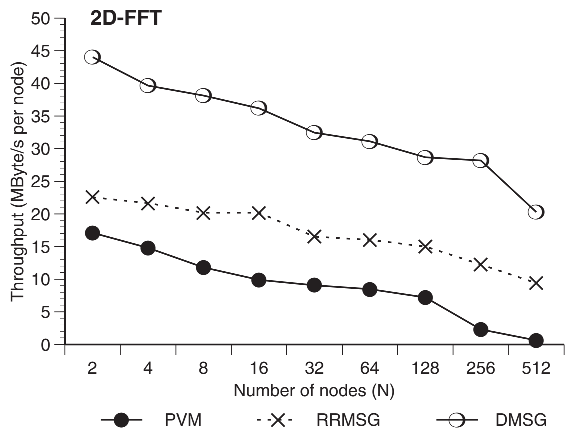


Figure 9.11: Per-node communication performance of a 4096×4096 point 2D-FFT for different T3D machine sizes.

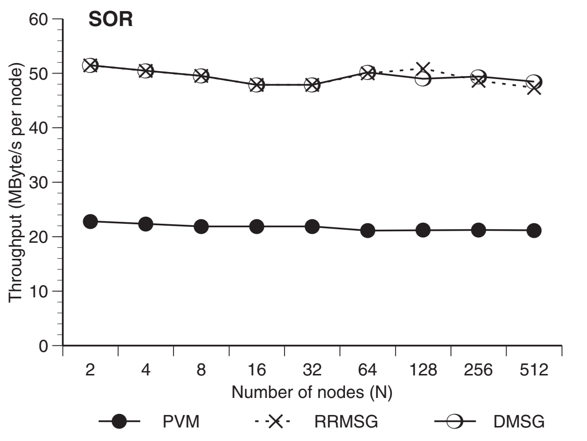


Figure 9.12: Per-node communication performance of a 4096×4096 element SOR for different T3D machine sizes.

limited even with large problem sizes and smaller machines. For larger machines, PVM seems completely limited by the constant per-message overhead spent on buffer allocation and the implicit synchronization of each data transfer. The overheads for control messages are also present in RRMSG, but the per-element overheads can be avoided, and the constant overhead seems smaller. Therefore the performance does not drop as fast as the number of nodes increases, resulting in improved scalability.

In DMSG I incorporate all advantages of decoupled synchronization. I use hardware barriers, direct deposit for data transfers, and additional barriers for congestion control in the network. Direct deposit eliminates buffering as well as the gather/scatter copies. With fast barriers on the T3D, there is minimal synchronization overhead, and high message transfer rates make application performance scale up without loss as more processing nodes are added.

The differences in communication performance are less pronounced for the SOR application kernel, which uses a simple next neighbor pattern and transfers large contiguous blocks of memory, but I still note significant performance differences. The reduced synchronization overheads of DMSG and RRMSG are less visible, but with PVM, the data is still copied several times due to the standard interface to the library. While DMSG and RRMSG achieve throughput numbers from 48 to 50 MByte/s, PVM is limited to 20 MByte/s, due to copying.

At transfer rates of 20 MByte/s per node, there is no network congestion for all-to-all communication on a T3D, except for the basic reduction of the link speed by a factor of two.[2] Despite the higher transfer rates of DMSG, the congestion in the router can be minimized with additional barriers for machine sizes up to 1024 according to a method described in [56]. Congestion control is impractical for PVM and RRMSG because the protocol messages synchronize each sender and receiver pair independently rather than with a global barrier.

9.1.5 Detailed analysis of communication time

To understand the full impact of the message passing style on the performance of my applications, I must quantify the times spent in communication-related work in more detail and investigate whether performance is lost through constant per-message overhead (e.g., startup or protocol overheads) or through linear per-byte cost (e.g., copies during buffering). To see the impact of data copies in the different styles, I examine *large* problem sizes in particular. I expect the constant per-message overheads to be more more visible at *small* problem sizes.

Figures 9.14, 9.15, 9.16, and 9.17 show the fraction of time my application kernels spend in computation and in communication. On the left, each figure depicts the total execution time in seconds. On the right, the communication time is further broken down into: *data transfer*, the time to actually transfer the data across the network; *barriers/control*, the time spent in synchronization; *gather/scatter*, the time required to gather all data into one contiguous block and scatter it into its final location; and *pack/unpack*, the fraction of time PVM spends to prepare the messages (pvm_fpack, pvm_funpack calls). Figure 9.13 illustrates this process for PVM on the T3D.

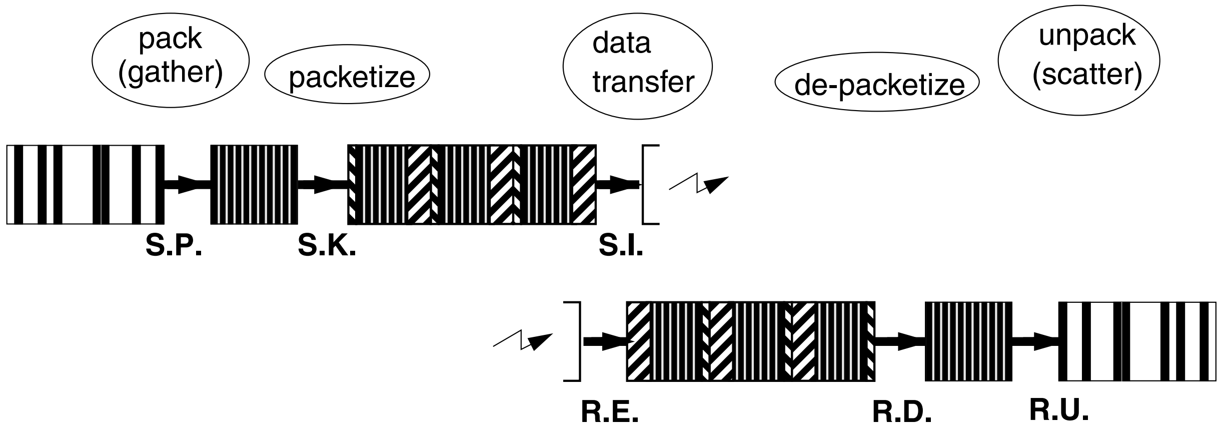


Figure 9.13: Possible copies of data occurring due to buffering for synchronization and coalescing of non-contiguous data transfers found in PVM a traditional message passing system.

[2]This reduction is due to the node architecture of the T3D. A single network access point is shared among a pair of processing nodes.

2D-FFT: large problem size

As Figure 9.14 indicates, for the larger problem size of 2D-FFT, the DMSG communication is almost a factor of fmy faster than PVM. The RRMSG case without hardware barriers is still about a factor of two faster than PVM.

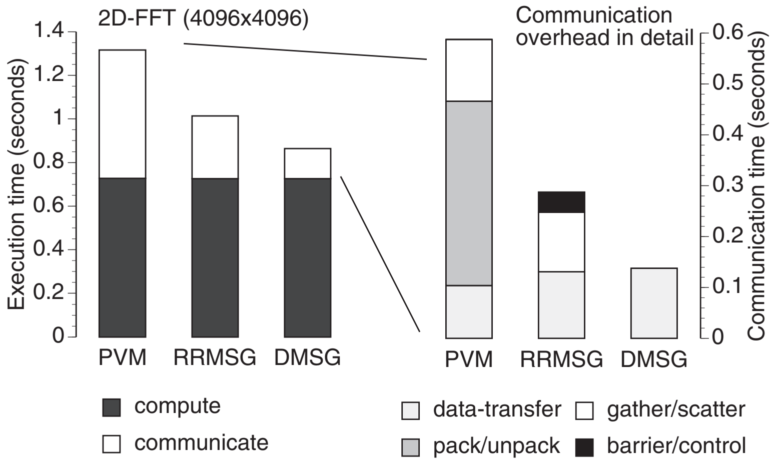


Figure 9.14: 2D-FFT (4096 $\times$ 4096) for different message passing models: detailed breakdown of execution time [128 nodes].

The amount of time spent doing the actual transfer of data across the network is virtually the same in all cases. The data transfer part of the DMSG library is somewhat slower since it includes the gather/scatter step, depositing strided data directly to its final destination. The RRMSG and PVM cases pay a significant additional cost to separate the two data transfer and gather/scatter. Furthermore, in DMSG and RRMSG, all buffer management can be done by the compiler, thus incurring no additional overhead during runtime. For PVM, the measurements quantify the cost of buffer management and copy overheads in the pack/unpack times.

The amount of time spent in synchronization and protocol processing is so small for DMSG, which uses fast hardware barriers for synchronization, that it is not visible in the figure. The time is larger for PVM, but because the PVM data transfer functions are integrated, I cannot separate the synchronization/protocol costs from the pack/unpack costs. The RRMSG synchronization cost includes the cost of a control message that synchronizes every data transfer in advance, which adds up to much more time than a single barrier.

SOR: large problem size

Not all applications have dense and communication-intensive patterns like the transpose in 2D-FFT. In some applications, the nodes just exchange an overlapping region of data with their immediate neighbors; SOR is such an application. With very few messages exchanged in the large SOR case, the benefits of decoupled synchronization and fast barriers come indirectly through less copying rather than directly through elimination of overheads, as depicted in Figure 9.15.

While DMSG and RRMSG transfer the contiguous blocks of data end-to-end without copying, PVM does not have enough synchronization information to store data directly into its destination (i.e., it cannot risk overwriting live data), and it seems to make at least one copy to an intermediate buffer. This is visible

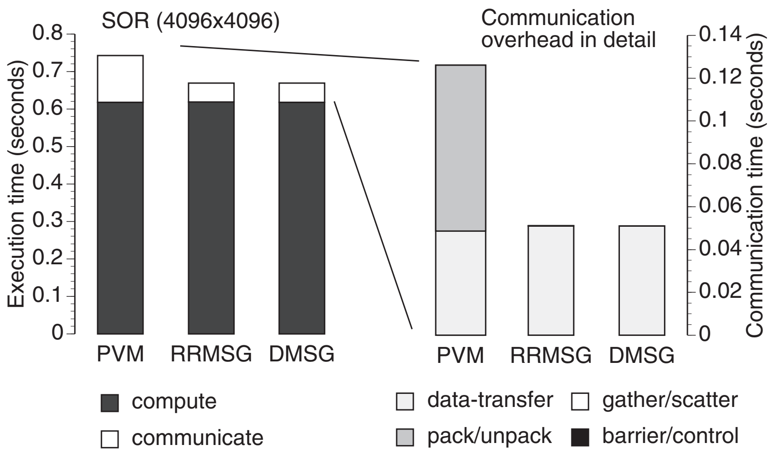


Figure 9.15: SOR (4096×4096) for different message passing models: detailed breakdown of execution time [128 nodes].

in the measured overall communication time. For the vendor implementation of PVM, the measured breakdown into transfer time and buffering overhead in the graph is meaningless. The measurements indicate that some of the actual data transfers and buffering are delayed until the pvm_unpack calls are made.

2D-FFT: small problem size

Figure 9.16 shows the 2D-FFT performance on a small problem size (256×256). Because of the large number of small messages, the constant per-message overhead for synchronization protocol and buffer management dominates the PVM and RRMSG times. With a relatively small amount of computation to do, the overall performance of the 2D-FFT is critically dependent on the message passing system used: DMSG is about 30 times faster than PVM, while RRMSG outperforms PVM by a factor of about 10.

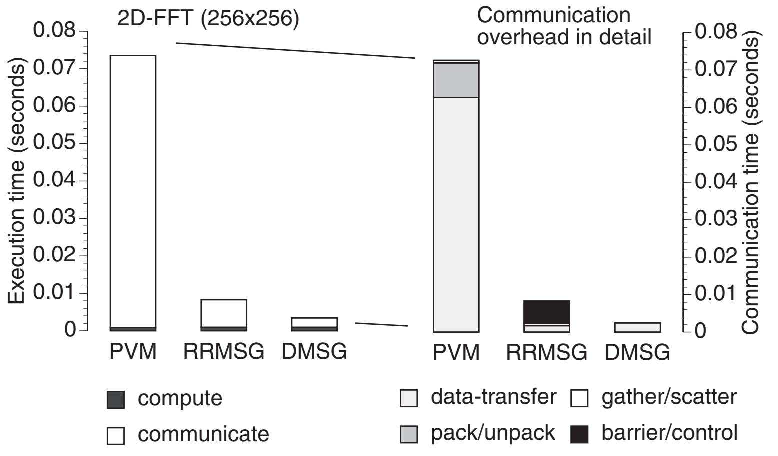


Figure 9.16: 2D-FFT (256×256) for different message passing models: detailed breakdown of execution time [128 nodes].

PVM spends most of its communication time in the basic data-transfer routines pvm_send and pvm_receive, which include both synchronization and actual data transfer. The extra cost of scattering/gathering and buffer packing is relatively small but visible. RRMSG incurs a significant synchronization overhead, because synchronization with control messages is quite costly in dense communication patterns. DMSG is faster because it relies on a global synchronization with fast hardware barriers; even with a small 2D-FFT problem size, the time of the barriers is not visible.

SOR: small problem size

In SOR, each node exchanges data only with two neighbors. Therefore, in comparison to the 2D-FFT communication pattern, I expect to see a reduced impact of per-message synchronization overhead, but a bigger impact of extra copies in PVM, which always buffers data. Figure 9.16 illustrates the SOR performance for a small (256×256) problem size.

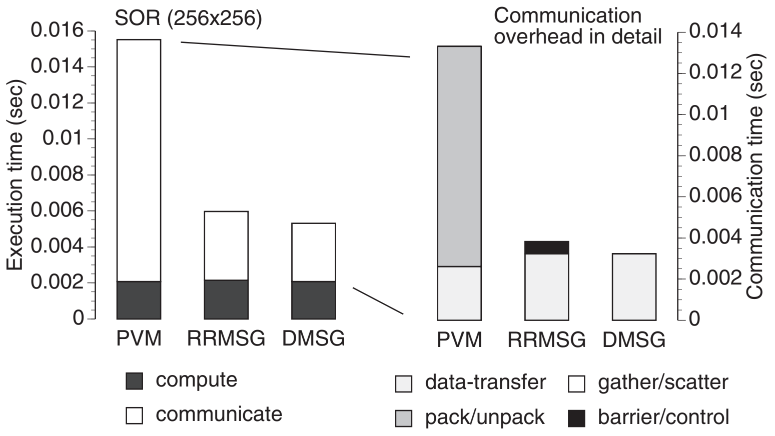


Figure 9.17: SOR (256×256) for different message passing models: detailed breakdown of execution time [128 nodes].

For SOR with smaller problem sizes, PVM seems to transfer data a bit faster to its internal buffers than DMSG and RRMSG can transfer data end-to-end. However, PVM incurs a large overhead due to internal copying triggered by the pvm_pack and pvm_unpack calls. For simple, sparse patterns like SOR, synchronization can be done with control messages at a reasonable extra cost on machines that do not have built-in barriers. Thus the extra synchronization overhead in RRMSG is small compared to the overall communication time.

9.1.6 Summary of SOR and FFT

For each application (SOR and FFT) and all three communication libraries (PVM, RRMSG and DMSG), the amount of data transferred is identical regardless of the message passing library used. I can therefore compare the time spent in communication to explain why DMSG has the highest transfer rates and the lowest communication overhead in both applications with both problem sizes. The scalability curves for my application kernels in Figures 9.9 and 9.10 follow from the characteristic overheads of the different message passing libraries in Figures 9.14 and Figures 9.15.

9.1.7 Optimization of an FEM code under MPI

Direct deposit compiler technology is applied successfully to optimize communication in the Archimedes compiler. Archimedes comprises the mesh generators, the partitioner and a parallelizing application generator used a NSF grand challenge code for earthquake simulation studied by my colleagues in computer science and civil engineering [89, 35, 6]. Unfortunately the balance between communication work and computation for the huge simulation problems studied a that grand challenge is such local computation dominates the application performance at this time. So portability, and with it the opportunity to use the parallel machine with the latest and most recent microprocessor, becomes more important than excellent communication efficiency. Portability was given priority over communication efficiency and after an initial prototype implementation based on iWarp deposit messaging, the Archimedes prototype was re-engineered for portable inter-processor communication using MPI so it could meet the requirements of the large meshes of earth quake modeling. An earlier solver used for a comparison between message-based and connection oriented communication in [36] uses the other variant of DMSG messaging with interrupt driven senders.

The MPI-implementation evolved from an initial version based on blocking send/receive calls into an optimized version that decouples synchronization and data transfer. Barriers are used for synchronization and split receives calls eliminated some intermediate buffering and unnecessary copies.

A preliminary code uses a communication schedule with a strict "all sends before receives" policy. The following schedule is executed in every step of the iterative solver:

S1

```
for (destination=0; destination < npes; destination++)
  send(destination,data);
for (source=0; source < npes; source++)
  receive(source,data_pointer);
```

The communication schedule S1 requires a messaging library that implements a clean postal model and requires buffer space to hold to whole data of a communication step. An improved, but still a traditional MPI implementation modified, applies a skewed schedule to reduce the required buffer space significantly at the cost of artificially sequentializing some messages, that could otherwise be transmitted in parallel. In schedule S2 each node first sends the messages to all other nodes with node_ids smaller than its own id, then it receives all messages it needs to receive and finally send the messages to nodes with larger node_ids [3]. The conventional implementation with skewed schedule still follows a postal model and relies on the internal buffering services of the library.

S2

```
for (dest=0; dest < my_cellid; dest++)
  send(dest,data);
for (source=0; source < npes; source++)
  receive(source,data_pointer);
for (dest=my_cellid; dest < npes; dest++)
  send(dest,data);
```

The communication performance with both of these implementations turns out to be non-satisfactory. An analysis of the mesh partitioning methods in the solver reveals that the fraction of the communication

[3]For the reduction of buffer-space this method requires a particular combination of a cell numbering scheme with a router that generates routes along paths with strictly increasing or strictly decreasing numbers. The common e-cube routers for meshes and hypercubes meet this requirement. This issue becomes a question of correctness once I attempt to reduce the limit on the maximal buffer space available.

volume is asymptotically decreasing for large problem and that therfore communication efficiency is not critical for this application. Still, the bad communication efficiency of the baseline MPI implementation meant a unacceptable slowdown of the application code due to slow communication in all interesting problem sizes, including the one second interval meshes containing hundred thousands of finite elements. Portability requires that even a communication optimized version maintains compatibility with MPI, ruling out a switch to direct deposit messaging.

The MPI standard includes separate synchronization primitives among its collective communication routines, and once a program is rewritten accordingly, synchronization can be decoupled from data transfer and be used to eliminate buffering completely for data messages even without sequentializing the communication schedule. The communication schedule revised MPI program (S3) follows the deposit model although no direct deposit mechanism is used.

S3

```
for (dest=0; dest < npes; dest++)
  split_rcv_start(dest,data_ptr,completion_semaphore);
barrier();
for (source=0; source < npes; source++)
  send(source,data);
for (dest=0; dest < npes; dest++)
  split_rcv_wait(dest,completion_semaphore);
```

The comparison of performance in the Figure 9.18 and 9.19 compare the decoupled MPI program to a conventional implementation of the same finite element solver. Two properties characterize each pair of execution times. The mesh resolution (in seconds) indicates the size of the mesh and therefore the problem size. A resolution of 5s translates into 30k points, a resolution of 2s into 380k points, and a resolution of 1s into 2.4 million points. The second number is the number of processors used in the run of that application — either 64 or 128 processors. Due to memory requirements the 1s meshes require at least 64 processor. The conventional implementation uses a skewed schedule to limit buffer space. Without such an optimization, large problem sets could not run in that particular MPI implementation, since MPI specification does not provide a general guarantee of a minimal system buffering space.

Figure 9.18 show a relative breakdown of the combined execution time into communication and computation work for both the conventional and the decoupled MPI implementation. Figure 9.19 contains the absolute numbers to put the relative percentages into perspective. The graphs in both figures emphasize the difference between the conventional MPI program and the decoupled MPI program. For large simulation problems and small machines the per byte costs in FEM communication are important. The decoupling optimization is most effective in those cases, and the fraction of communication is reduced from 30% to about 6% for the 1s mesh running on 64 processors. For small problems the individual messages are too small to become optimized by decoupled communication and the constant per message cost of MPI dominates the communication performance completely. Switching to the simpler direct deposit messaging mechanism (or alternatively to Crays' shmem_put library) would help to speed up those cases, but jeopardize the portability of the application.

The percentage of execution time in communication clearly indicate the trend to lower communication costs in the deposit model and the effectiveness of the optimization. The absolute timings may not indicate this trend so clearly. The un-optimized and the optimized numbers were gathered from the same partitioned meshes but some slightly different levels of optimizations were used in the solver as it evolved. Also, the time for I/O operations is excluded, partly because the code I/O for operation was rewritten in between the two versions and partly because I/O performance measurements remain irreproducible on the T3D at the PSC. On this T3D all I/O is performed by the front end, a heavily loaded multi-headed C90 vector

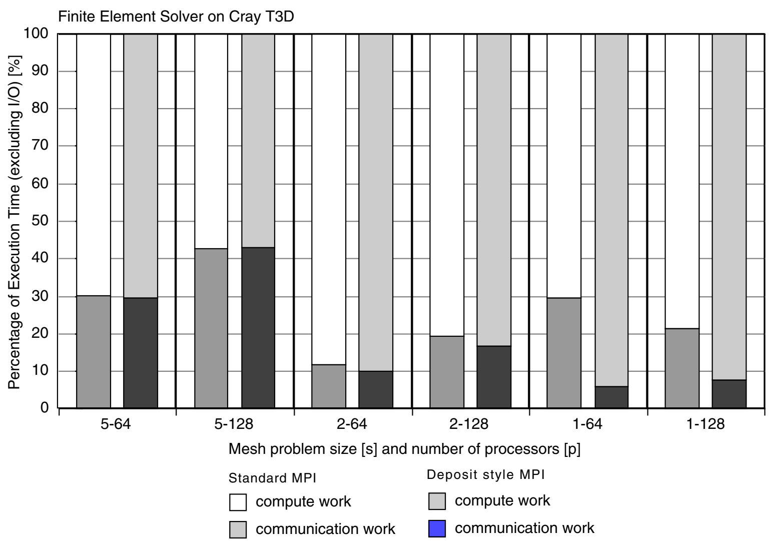


Figure 9.18: Fraction of execution time spent in communication for two MPI implementations of FEM (earthquake model) on a T3D. Left bar: a conventional postal message passing program and right bar: a deposit model program with split receives.

processor and main work-horse of the PSC.

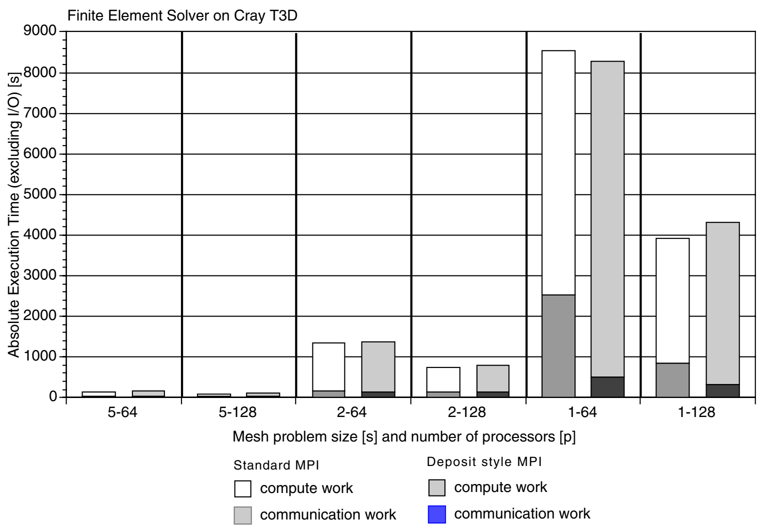


Figure 9.19: Execution times for two MPI implementations of FEM (earthquake model) on a T3D. Left bar: a conventional postal message passing program and right bar: a deposit model program with split receives.

With adequate support direct deposit can handle indexes accesses almost as well as strided accesses and further optimized the decoupled MPI program. The iWarp, T3D, and Paragon can transfers address data pairs to deposit strided data directly to its final destination. A set of micro-benchmarks use the sparse access pattern of an early 20k earthquake model mesh and simulate the access of the solver. The performance data of FEM with direct deposit in Section 9.2.7 are based on those measurements and reflect all architectural consideration for direct deposit communication with indexed access patterns. It just remains beyond the scope of this architectural investigation to integrate these mechanisms into a production quality FEM-solver like the grand challenge quake code. An integration of those technologies is planned since both Archimedes (FEM compiler) and Fx (HPF compiler) will use the Catacomb code generator at some time in the future.

9.2 Optimizations of data transfers

A second experimental framework serves as a showcase for a memory system based optimization of data transfers. Two parallel supercomputers, a Cray T3D and an Intel Paragon, are used to investigate the best way to perform communication for applications with transposes and exchanges of indexed arrays. The best direct deposit mechanism (chained memory operations) is compared to the conventional buffer packing approach used by all standard libraries that are limited to contiguous blocks (e.g., PVM, MPI and early versions of SHMEM_PUT). The section starts with the necessary micro benchmarks for a local and remote memory system performance characterization and ends with the actually measured performance numbers for data transfers within the three application kernels. Note that the FEM numbers in this framework are based on a synthetic kernel based on an early 2.5 dimensional earthquake mesh (q20k, 20000 points).

9.2.1 Throughput of local copies

The throughput for the basic local memory-to-memory transfers ${}_xC_y$ critically depends an the access pattern as seen in Table 9.2.

	$\vert{}_1C_1\vert$	$\vert{}_1C_{64}\vert$	$\vert{}_{64}C_1\vert$	$\vert{}_1C_\omega\vert$	$\vert{}_\omega C_1\vert$
T3D	93	67.9	33.3	38.5	32.9
Paragon	67.6	27.6	31.1	35.2	45.1

Table 9.2: Throughput of selected local memory-to-memory transfers (MB/s) for large blocks.

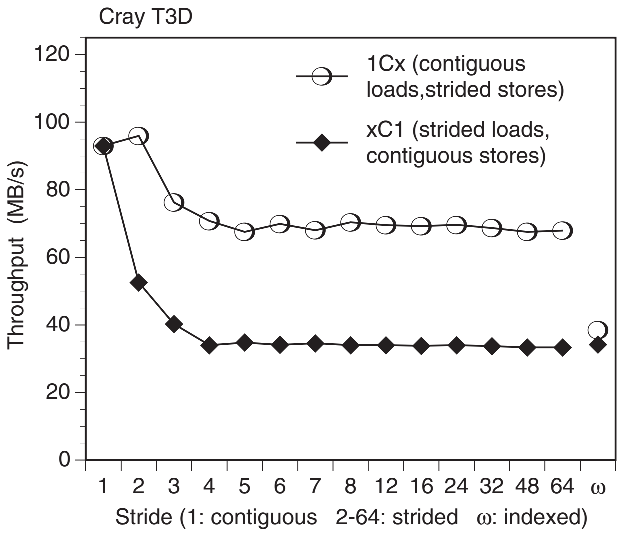
The graph in Figure 9.20 show the different characteristics of the memory systems on T3D and Paragon, when strides are involved. On the T3D strided stores are better supported because of the write back queue. On the Paragon strided loads can be pipelined and benefit from the pre-fetch queue.

9.2.2 Throughput of send/receive copies

The throughput for the network access depends partly on local memory-to-memory transfer and partly on network limitations. The measured figures are given in Tables 9.3 and 9.4. Since the numbers do not vary for large strides, I assume for simplicity that the throughput for stride 64 applies to any larger stride.

	$\vert{}_1S_0\vert$	$\vert{}_1F_0\vert$	$\vert{}_{64}S_0\vert$	$\vert{}_\omega S_0\vert$
T3D	126	-	35	32
Paragon	52	160	42	36

Table 9.3: Throughput figures for sending network transfers (MB/s).



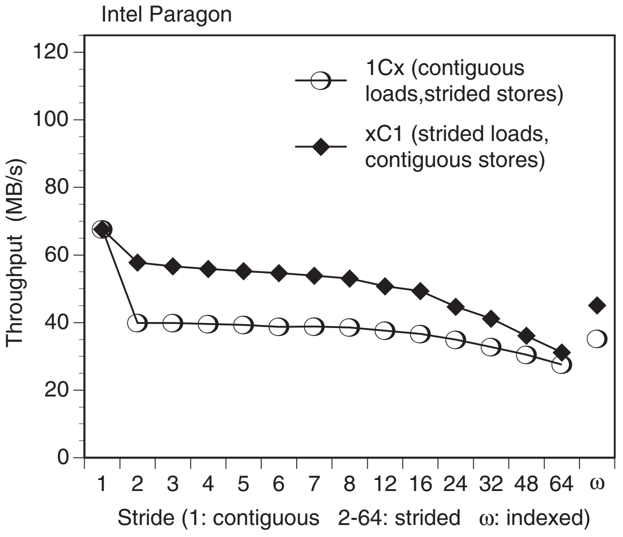


Figure 9.20: Throughput for strided local memory-to-memory transfers (MB/s).

	$\vert_0R_1\vert$	$\vert_0D_1\vert$	$\vert_0R_{64}\vert$	$\vert_0D_{64}\vert$	$\vert_0R_\omega\vert$	$\vert_0D_\omega\vert$
T3D	-	142	-	52	-	52
Paragon	82	160	38	-	42	-

Table 9.4: Throughput figures for receiving network transfers (MB/s).

9.2.3 Congestion and throughput of the network

Network congestion is *absent* from my model. This may seem surprising at first, since none of the machines of interest to us provides a fully scalable bisection bandwidth as, e.g., the CM-5[71]. Both machines use a simple mesh topology with fast links for their communication networks. In my experience, the raw link speed in the network significantly exceeds the effective throughput achievable in useful data transfers. For most applications, the machines will not be network-congestion limited unless I move to very large machines. There are however two quirks: On the T3D, two adjacent nodes share a single communication port to the network. This design feature introduces congestion at the access point, and therefore the minimal congestion is *two* unless half of the processors remain unused. For the Paragon, the unfortunate aspect ratio of certain machine sizes (e.g., 112x16) and the lack of torus links can cause congestion for some patterns. In general, next neighbor patterns like cyclic shifts cause just a small congestion of one or two, and even dense patterns like the complete exchange or personalized all-to-all communication can be scheduled with minimal congestion on T3D tori of up to 1024 (2x8x8x8) compute nodes[55].

Because of these two problems in the T3D and Paragon networks, communication runs at a congestion of two in many cases, and I use the measured throughput for this congestion, when using my model to compute overall throughput. For completeness, Table 9.5 shows network performance at congestion one, two, and four. Congestion two means a network link is traversed by twice as much data as it can support at peak speed. For a throughput oriented model it is irrelevant whether the data are multiplexed at a per flit or a per message level.

For the network throughput, it is more important whether just the data words are transferred, or if the addresses for remote stores are transferred along with the data words (address-data pairs). I have therefore measured the network bandwidth for large block transfers for both options (data only and address-data pairs) for different fixed congestion factors. The bold data in Table 9.5 indicate what I consider to be representative values for my class of applications.

	Average congestion					
	data only (N_d)			address data pairs (N_{adp})		
	1	**2**	4	1	**2**	4
T3D	142	**69**	35	62	**38**	20
Paragon	176	**90**	44	88	**45**	22

Table 9.5: Network bandwidth (MB/s) as a function of a fixed overall congestion.

9.2.4 Optimization of communication operations

The large variety of access patterns and hardware capabilities implies that there are different ways to implement a particular communication operation ${}_xQ_y$ by composing it out of different basic transfers. Looking at the Cray T3D and the Intel Paragon, I identify different tradeoffs in the design of the most important communication operations of a parallelizing compiler. In both cases the copy-transfer model guides an optimization towards maximal performance.

9.2.5 Buffer-packing vs. chained transfers

Section 6.2 presents an example of buffer packing, but with appropriate hardware support, the buffer packing/unpacking copy steps can be eliminated. That is, I can implement the communication operations ${}_xQ_y$ for the T3D and the Paragon so that they avoid packing buffer(s). These implementations (and their bandwidths) are different for the two machines, but the overall idea is the same. Therefore, a compiler or user has two options when selecting communication operations to perform a computation step:

Buffer-packing transfers The buffer packing message passing libraries (such as PVM) attempt to transfer contiguous blocks at all costs, leaving the packing / unpacking of communication buffers to the application code. Packing and unpacking is done through a local copy in memory before and after the transfer across the network. This arrangement benefits from *faster* contiguous transfers across the network but suffers from the cost of additional accesses to local memory. Figures 9.21 and 9.22 illustrate the path of data for this style (of course, these operations are overlapped or pipelined as stated in Section 6.2.4).

Chained transfers By chaining the *slower* non-contiguous accesses to data with the transfer of data from local memory to the network at the sender side (and vice versa for the receiver side), I eliminate local copies at the expense of supplying the data more slowly to the network. The chained transfers rely on the deposit engine at the receiver node to perform the stores. Figures 9.23 and 9.24 illustrate the flow of data within a node.

The flexibility of chained transfers with strided and indexed memory accesses occurs at a cost. Transfers with these patterns are expected to be slower than contiguous block transfers as my measurements indicate and my performance parameters take into account. This is partly due to the work of gathering and scattering strided data and partly due to the loss of specific hardware support when patterns become more complex. Remember that the access pattern of DMAs and other dedicated hardware is often restricted to contiguous transfers.

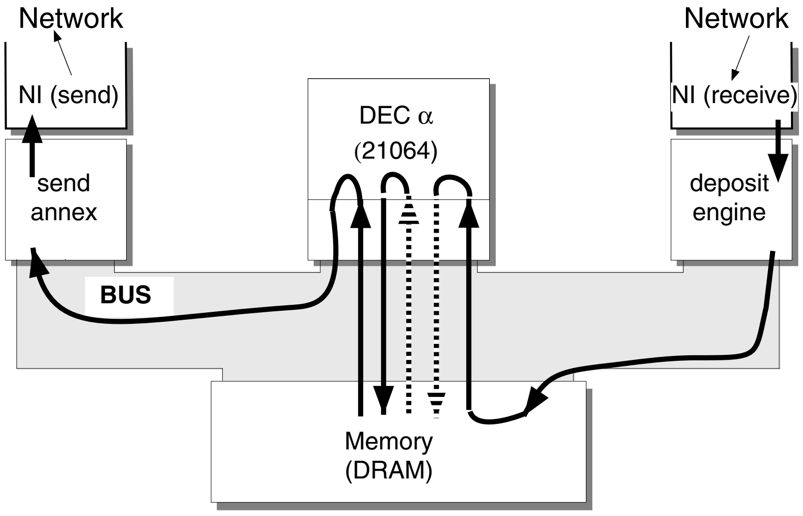


Figure 9.21: Schematic flow of data for buffer packing communication in a Cray T3D node. Solid lines indicate streams of contiguous data, dashed lines potentially strided or indexed data.

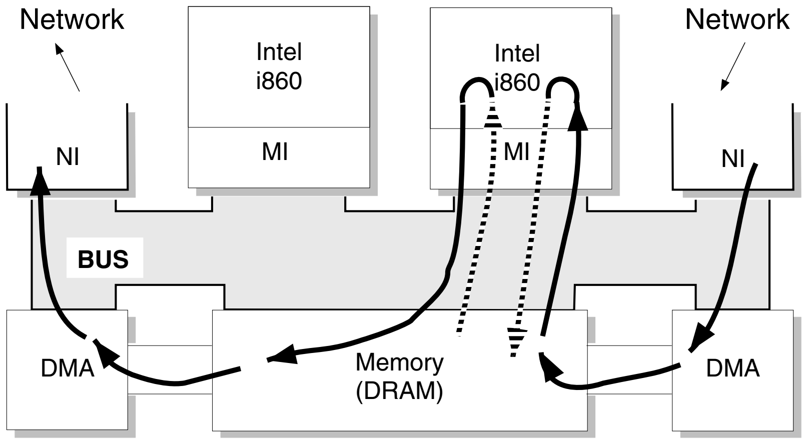


Figure 9.22: Schematic flow of data for buffer packing communication in an Intel Paragon node. Solid lines indicate streams of contiguous data, dashed lines potentially strided or indexed data.

Counting the number of transfers from and to the memory system for each case, it becomes evident that the chained communication results in less copying and therefore in a lower requirement for memory system bandwidth. However, counting the accesses does not take into account the variation of memory system bandwidth due to different access patterns in each basic transfer to and from memory.

Buffer packing transfers on the T3D

In the Section 6.2.4 I presented the formula for buffer-packing message passing. This message passing style is provided by both the Cray PVM library on a higher level and the Cray SH_MEMPUT library

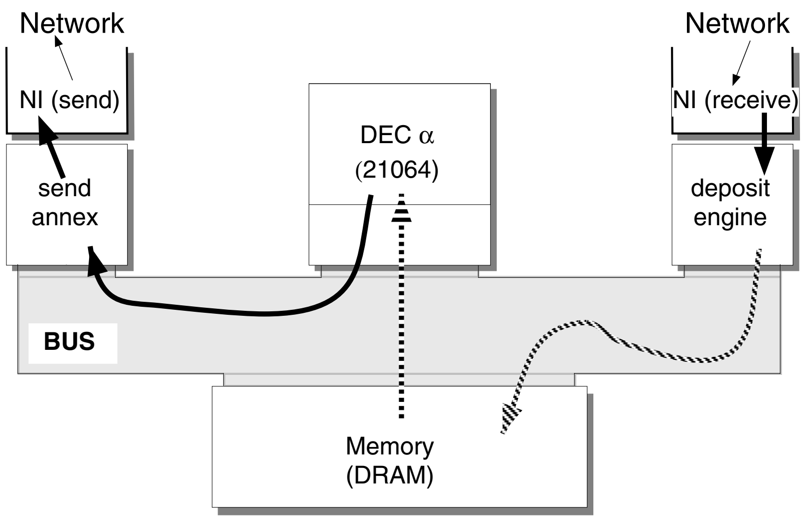


Figure 9.23: Schematic flow of data for chained communication in a Cray T3D node. Solid lines indicate streams of contiguous data, dashed lines potentially strided or indexed data.

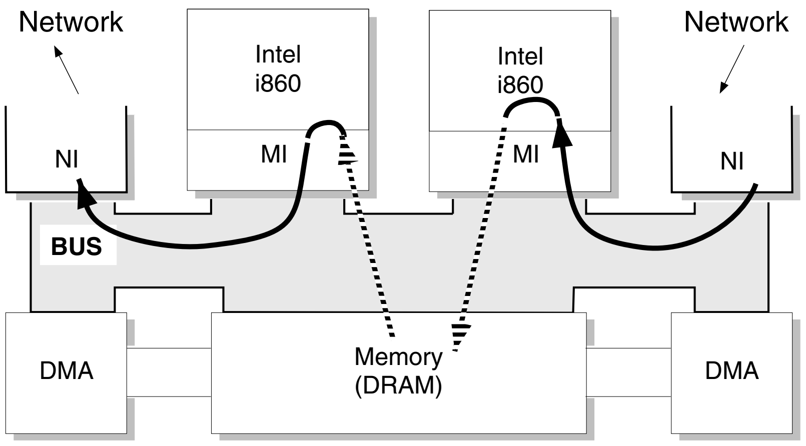


Figure 9.24: Schematic flow of data for chained communication in an Intel Paragon node. Solid lines indicate streams of contiguous data, dashed lines potentially strided or indexed data.

(libsma.a) on a lower level. While both libraries contain primitives for direct contiguous memory transfers, both libraries fail to provide adequate direct transfers for strided and indexed transfers without local copying in memory. Furthermore, the performance of PVM is reduced by additional copies to temporary system buffers

The buffer packing message passing primitive $({}_xQ_y)$ on the T3D is implemented as composition of the following basic transfer steps:

$$ {}_xQ_y \; = \; {}_xC_1 \circ ({}_1S_0 \| N_d \| {}_0D_1) \circ_1 C_y $$

Using the model of Section 6.2, I obtain these performance estimates:

$$|{}_1Q_1| = 27.9 \text{ MB/s} \quad |{}_1Q_{64}| = 25.2 \text{ MB/s}$$
$$|{}_{64}Q_1| = 17.1 \text{ MB/s} \quad |{}_\omega Q_\omega| = 14.2 \text{ MB/s}$$

The T3D offers hardware support to perform direct user-space to user-space transfers for all communication patterns: contiguous, strided, and indexed. This capability potentially eliminates the buffer packing at the sender and unpacking at the receiver end even for the more complex access patterns, at the cost of possibly slowing down the network transfers.

Chained transfers on the T3D

A chained implementation ${}_xQ'_y$ of the basic inter-node transfer avoids the local copying steps. On the T3D, such an implementation must be done at the (dis-)assembler level, and although this approach is too tedious for a programmer, it may be appropriate for a compiler. Also, a better user interface to the annex hardware could alleviate some problems. The chained implementation ${}_xQ'_y$ exploits the flexibility of the deposit engine to handle all access patterns, including strided and indexed accesses. Using my basic transfer steps, I have two cases:

$$\begin{aligned} {}_1Q'_1 &= {}_1S_0 \| N_d \| {}_0D_1 \\ {}_xQ'_y &= {}_xS_0 \| N_{adp} \| {}_0D_y \end{aligned}$$

Using the concatenation rules of Section 6.2.4, my model predicts:

$$|{}_1Q'_1| = 70 \text{ MB/s} \quad |{}_1Q'_{64}| = 38 \text{ MB/s}$$
$$|{}_{16}Q'_{64}| = 38 \text{ MB/s} \quad |{}_\omega Q'_\omega| = 32 \text{ MB/s}$$

Figure 9.25 shows measured throughput rates for buffer packing and chained transfers on the Cray T3D and the Intel Paragon, for different access patterns. As can be seen, the model predictions match fairly accurately the measured performance.

9.2.6 Tradeoff: Strided loads vs. strided stores

When implementing the communication primitive for a two dimensional array transpose, the compiler can choose between an access pattern of ${}_1Q_n$ or ${}_nQ_1$ in the remote memory transfer, as seen in Figure 9.26.

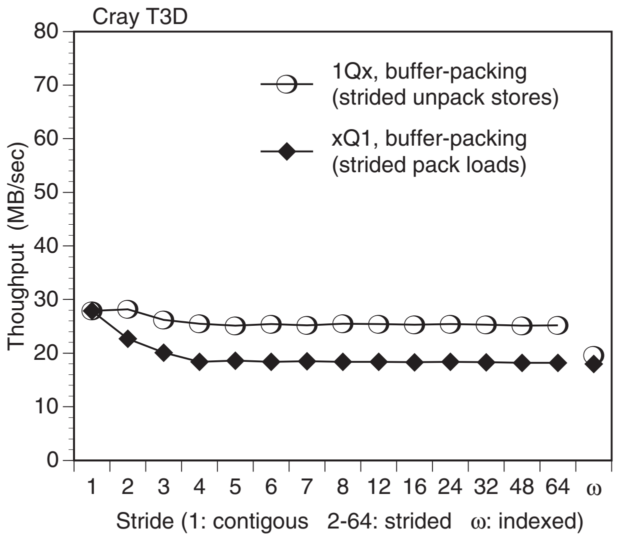
This choice corresponds to the (arbitrary) choice of `i` or `j` as an outer looping variable in a transpose loop with body `b[i][j]=a[j][i]`. Both implementation of this transfer are possible.

Using the bandwidth parameter rules of my copy-transfer model, the effective bandwidth of the communication operations is predicted as seen in the Table 9.6.

	T3D *model*		Paragon *model*		T3D *measured*		Paragon *measured*	
MB/s	Buffer packing	Chained	Buffer packing	Chained	Buffer packing	Chained	Buffer packing	Chained
${}_1Q_{16}$	25.4	38.0	18.3	32	20.8	31.3	20.7	29.7
${}_{16}Q_1$	18.4	38.0	20.7	42	14.3	27.4	24.2	39.2

Table 9.6: Estimated and measured performance for strided loads vs. strided stores.

This optimization of choosing strided stores on the T3D and strided loads on the Paragon is not surprising, given the better performance of strided stores for memory-to-memory copies in one architecture and



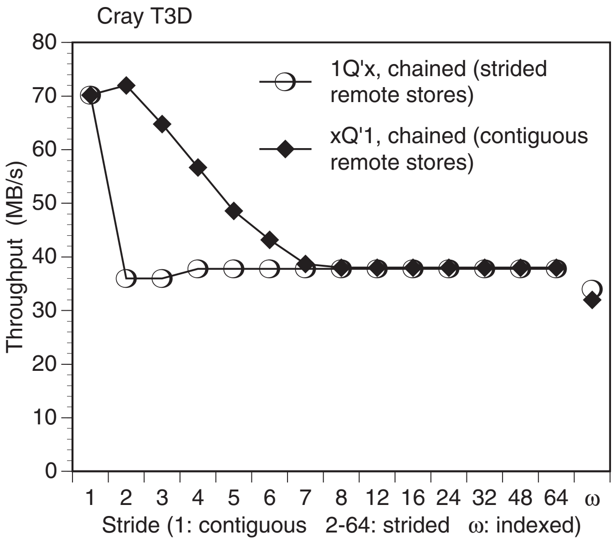


Figure 9.25: Throughput for communication operation with different strided access patterns including contiguous, strided and indexed for either loads or stores. The buffer-packing implementations (left) result in a lower throughput than the chained implementations (right).

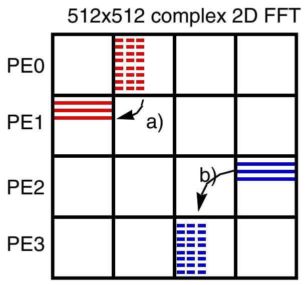


Figure 9.26: Execution of a 2D FFT includes an array transpose to change the distribution from row-major into column-major. Square patches must be moved between the processors. The patches of data can be moved in two ways, a) or b).

strided loads in the other architecture. Support for non-cacheable memory operations includes the write back queue of the T3D and the pre-fetch queue on the DEC Alpha and the pipelined loads of the Intel i860. I see that both of them improve communication performance over the equivalent cacheable memory operations, especially for complex indexed pattern. Unfortunately, the standard single-node compilers do not generate code for pipeline memory operation and the standard run time environment does not allow the user to use either write-back policy of the caches.

9.2.7 Measured performance in application kernels

To evaluate the appropriateness of the copy-transfer model for applications (and not just basic communication operations as discussed in Section 9.2.4), I choose the communication kernels of three important applications. Two of these applications are compiled by a compiler for my dialect of HPF and one by an application-specific compiler. They are run on the T3D (since it is easier for us to explore architectural aspects on this machine than on the Paragon). The three applications were chosen to observe representative communication patterns.

Application kernels

As in the first Framework the same three kernels used for my evaluation: an array transpose, as it occurs in 2D FFT, the communication of a solver step in a finite element method (FEM) program and the communication occurring in a successive over-relaxation (SOR) solver. However the problem sizes and the implementations can be slightly different. Here are the precise specification of each test case.

Transpose in 2D FFT

Transposes are important to many application. My example is taken from an $n \times n$ 2D FFT application kernel. I choose a 1024×1024 complex 2D FFT because I observed this problem size to be common for applications on this class of machines. The transposes are necessary to provide locality for the column FFTs after the row FFTs are completed. I encountered a transpose of similar size as the performance critical communication step of a grand challenge application in air-shed modeling [76]. This code redistributes a $3500 \times (35 \times 5)$ array between one phase that performs numerical chemistry calculations and another phase that calculates transport phenomena, and this redistribution is implemented as a generic transpose.

Iterative solver on partitioned Finite Element graph

The FEM application kernel is derived from a sparse system solver based on a partitioned finite element graph, representing a 3 dimensional model of an alluvial valley surrounded by hard rock. This graph is used by our colleagues to study earthquakes [89]. Since the structure is an irregular 2.5 dimensional well partitioned grid, only a small fraction of the local data elements is exchanged between nodes, and the communication involves indexed accesses with arbitrary strides.

Successive over-relaxation solver

Not all applications require the transfer of strided or indexed data. SOR methods distribute data as contiguous blocks. A common technique is to replicate and overlap a region between neighbor processors to allow the computation to span across node boundaries. After every computation (relaxation) step, the overlap region is exchanged, using a shift communication step. In this case, I deal with matrix of size 256×256.

Modeled and measured performance for application kernels

For each application kernel I determine the throughput of the communication step for both buffer-packing communication and chained communication. Table 9.7 shows the throughput estimate of my model as well as the actual measurement on a 64-node partition of a T3D.

To put the numbers in Table 9.7 into perspective: these figures are very good numbers for these applications on the T3D. Using the standard vendor supplied message passing system, the performance is significantly less. Due to the constant overhead for sending a message in standard message passing libraries like PVM, the buffer packing numbers decrease drastically if I use Cray PVM3 . The PVM3 application performance is approximately 2 MB/s for FEM, 6 MB/s for FFT, and 25 MB/s for SOR.

	Buffer-packing		Chained	
	measured	model	measured	model
Transpose	20.0	25.2	29.5	38.0
FEM	12.2	14.2	20.2	32.0
SOR	26.2	27.9	68.1	70.2

Table 9.7: Measured data transfer rates of my application on a 64-node partition of a 512-node T3D, (MB/s *per node*).

While the T3D code reaches described here reaches production quality and is incorporated into the Fx compiler back-end the Paragon code suffers too many problems from hardware bugs or performance anomalies and therefore the code remains purely experimental and the performance numbers suitable only for a limited use in architectural studies.

9.3 Coherent shared memory (SMP) vs. direct deposit messaging (MSG-PASS)

For the study and evaluation of different memory system designs and different communication styles in compiled parallel programs I coded the 2D-FFT application kernel in Fx Fortran and compiled it into Fortran 77 cell programs with our Fx compiler. Two different back-ends are used to generate communication code for either a DEC 8400, a symmetric coherent shared memory multiprocessor, or for a Cray T3D, a massively parallel distributed memory system with support for remote stores.

9.3.1 Implementation of 2D-FFT

Like all high performance multidimensional FFTs, the Fx code must achieve locality of reference in both computation steps, the row and column FFTs. Therefore my 2D-FFT has four characteristic steps: local row FFTs (1D), global row-column transpose, local column FFTs (1D), global column-row transpose. The transposes are indicated to the compiler by an assignment statement of two distributed arrays. The implementation operates on complex numbers represented as a pair of 64bit, double precision floating point numbers. Since Fx Fortran allows to call external functions, I can rely on the best available library routine for a local 1D-FFT. Both vendors provide such an optimized FFT routine as part of their scientific library packages.

9.3.2 2D-FFT overall application performance

The 2D-FFT application kernels are executed for different problem sizes on all three systems, the DEC 8400, the Cray T3D and the Cray T3E. (I have not been able to gather all data points for the T3E at this time.) The performance measurements on four processors give a good impression of the impact of local and remote memory systems performance on the overall performance of the 2D-FFT.

Looking at the performance of the Fx Fortran program in Figure 9.27 I see that the DEC 8400 delivers a slightly higher performance than the Cray T3D on all different problem sizes of the 2D-FFT. For a 256×256 point 2D-FFT the Cray has an overall performance of 133 MFlops with four processors while the DEC 8400 peaks with about 220 MFlops, an difference in performance of about 50%. This result is somewhat surprising, given that a processor of the DEC 8400 is clocked twice as fast, has 3 levels of caches, and can issue up to twice as many instructions per clock than the T3D processor.

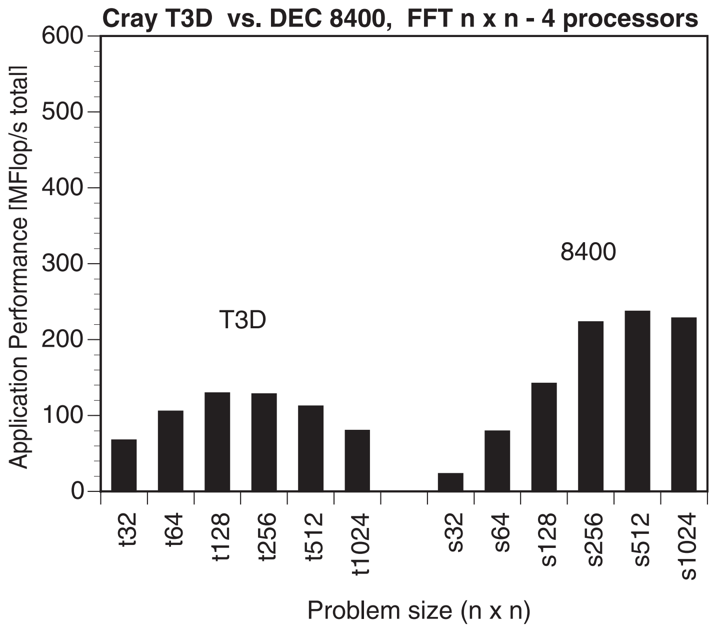


Figure 9.27: Total application performance of the 2D-FFT benchmark on 4 processors of a Cray T3D (left) and a DEC 8400 (right). Benchmark written in Fx Fortran.

9.3.3 2D-FFT computation and communication performance

For a more detailed analysis, Figures 9.28 and 9.29 depict the performance figures for computation and communication separately. For all problem sizes, the overall per processor performance depends on both factors, the local computation performance and the communication performance of the transpose. The parallel 2D-FFT application kernel is well balanced with regard to computation and communication work.

The computation performance in Figure 9.28 is strongly affected by the memory hierarchy and the cache structure of the processor node. From the graphed local computation performance numbers (in MFlops) it is evident that the sum of local computation performance over all four processors is more than a factor 2.5 higher on the DEC 8400 than on the Cray T3D. Looking specifically at large problem sizes for the 2D-FFT (e.g., 1024×1024, a million elements), I realize that the performance on the T3D falls off with large problems, while the performance on the DEC 8400 stays nearly at the same level. This performance advantage of the DEC 8400 is due to the higher clock frequency and the better caches in its memory hierarchy. The large L2 and L3 caches allow the DEC 8400 to execute the row and column FFTs out of cache — rather than out of DRAM memory — for the problem sizes above 256×256 elements.

The benefit of a faster processor design on the DEC 8400 is reduced by a communication system that runs at approximately the same performance level as the communication system on the Cray T3D. The better peak performance for local 1D-FFTs in Figure 9.28 is reduced by the communication overhead during the global transpose steps between row and column FFTs. To sustain the 2.5 fold performance improvement caused by its faster microprocessor and memory hierarchy, a similar performance improvement for the communication system would be required. This is not the case. The DEC 8400 offers only lower or similar bandwidth for communication in transposes, thus limiting the overall performance to a factor below two over the T3D. From the local and remote memory performance characterizations in Sections 8.4 and 8.5 I anticipate that the Cray T3E can fully utilize its better processor and communication system to

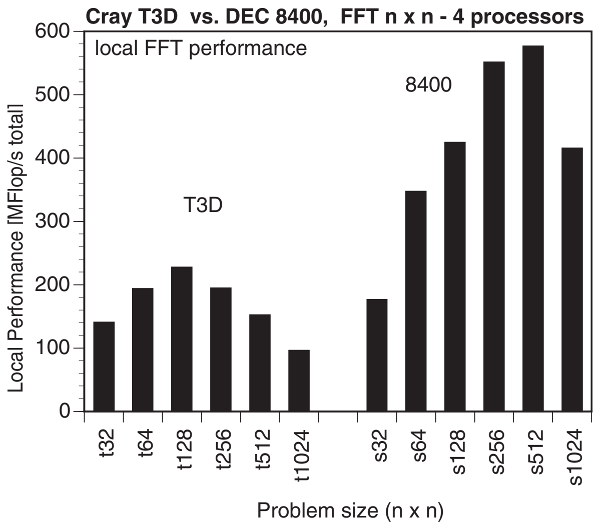


Figure 9.28: Local computation performance of the 2D-FFT benchmark on 4 processors of a Cray T3D (left) and a DEC 8400 (right). Benchmark written in Fx Fortran.

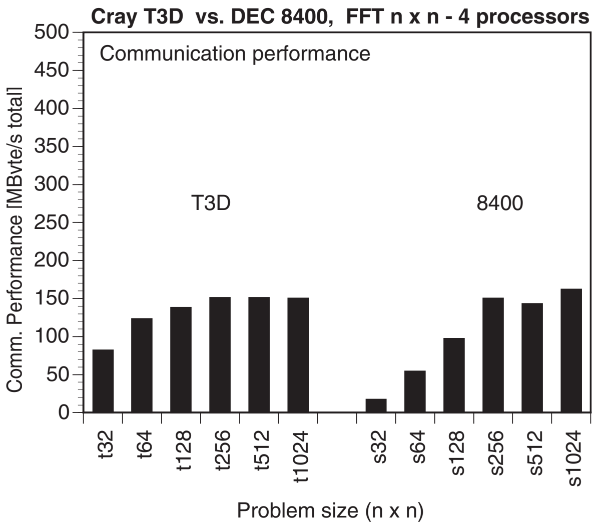


Figure 9.29: Communication performance in the transposes of the 2D-FFT benchmark on 4 processors of a Cray T3D (left) and a DEC 8400 (right). Benchmark written in Fx Fortran.

achieve a performance improvement of at least a factor of 3 over its predecessor, the Cray T3D.

9.3.4 Other factors

For my memory system characterization I focus solely on the performance of the processing nodes in both machines, with a particular emphasis on the performance of the local and remote memory system. I intentionally omit issues like ease of use, scalability of the network, I/O performance, or even cost of the systems in my comparison of the two machines. I acknowledge that these must be considered in an overall evaluation or a review of the two machines. During my experimental work with both machines I made a few observations concerning ease of use and scalability of the two systems that are worth mentioning.

Scalability in performance and cost

The Cray T3D is built as highly scalable machine. Parallel programs with just four processors are only "modestly" parallel for a Cray T3D, given the potential parallelism of executing them on a full size machine with 512 processors. On a T3D, the "massively" parallel performance of my compiler generated 2D-FFT written in Fx Fortran stays around 20 MFlops/s per processor and reaches a total performance of 8.75 GFlop/s when run on 512 processors. The code shows almost linear scalability from 16 to 512 nodes as seen in Section 9.1.

Scalability to a large number of processors was not a target for DEC 8400 series of machines and did not constrain the design. According to [31] a DEC8400 is limited to 12 processors and/or 14 GByte of memory. The maximal configurations are subject to the limitation of nine bus slots in the backplane with room for eight modules with either 2 CPUs or 2GByte of memory each.

Summary

Looking at my memory system characterization in Chapter 8 and the measured performance of a simple 2D-FFT application kernel, I find the overall application performance on a DEC 8400 is only about a factor of 1.5 higher, on average, than on a Cray T3D partition with the same number of processors. The benefit of a faster processor design on the DEC 8400 is reduced by a communication system that runs at approximately the same communication performance as the communication on the Cray T3D. The better peak performance for local 1D-FFTs in Figure 9.28 is reduced by the communication overhead during the global transpose steps between row and column FFTs. To sustain the 2.5 fold performance improvement by its faster microprocessor and memory hierarchy a similar performance improvement for the communication system would be required.

Based on the micro-benchmarks reported in Section 8 I anticipate that only on a Cray T3E, I will probably see the full performance gain of the second generation Alpha microprocessor once a production machine is installed at the Pittsburgh Supercomputer Center and is is available for use.

There are factors other than performance that matter in the evaluation of a system. The ease of use (full coherency) and better entry level costs for bus based symmetric multiprocessors are certainly appealing, but in this case I see that such advantages come as a tradeoff against memory system performance. I managed to re-target our parallelizing Fx compiler within an afternoon to the DEC 8400, while it took considerably longer to come up with a good T3D back-end for Fx, and I continue to refine a T3E back-end.

9.4 Simple test cases vs. the compiler generated tests

The third evaluation framework demonstrates remote memory systems performance, an interesting computer architecture design issue, in a compiler generated parallel program. The performance differences of a bus based SMP vs. the performance of network based supercomputer is properly modeled by micro-benchmarks in Chapter 6 and can be verified based on timings gathered with a Fx program. In the two previous frameworks a code fragment was extracted from an Fx program and examined as an isolated piece to keep the experiment as simple as possible. This is not a big problem since the extracted piece of code follows exactly the compilers' coding model, even when it is modified and optimized by hand. The computation part of my application kernels consists of highly optimized vendor libraries (libsci.a) and both parallelizing Fx compilation systems produces C-code to generate the communication code.

As for any tool-chain used in high performance computation, there are a few subtleties that change performance for the better or the worse (e.g., the cache alignment of arrays may be different, depending on which pragmas or linker options are invoked). The test application of the last framework, the compiler generated 2D-FFT is runs in both environments, the full Fx compiler environment (input in Fortran Fx, no modification to any intermediate codes) and simplified evaluation environment based on a all C code fragment.

9.4.1 Implementation of 2D-FFT in C and in Fortran Fx

For benchmarking, I coded and fine-tuned a 2D-FFT application kernel in C. This "best possible" implementation serves as my reference case. To test the performance with compiler generated parallel programs I code the same 2D-FFT application kernel in Fortran Fx and compiled it into Fortran 77 cell programs with my Fx compiler. The FFTs have four characteristic steps: local row FFTs (1D), global row-column transpose, local column FFTs (1D), global column-row transpose. The transposes are indicated to the compiler by an assignment statement of two distributed arrays.

Both implementations, the Fx Fortran and the hand-coded C programs operate on exactly the same data type, a complex number represented a as a pair of 64bit, double precision floating point numbers. Since Fx Fortran allows to call external functions, I can rely on the best available library routine for a local 1D-FFT and therefore eliminate most differences due to differences between the C and the Fortran compiler as such. Both vendors provide such an optimized FFT routine as part of their scientific library packages.

Comparison of overall application performance

I run my 2D-FFT application kernels written in C and Fx Fortran with different problem sizes on both systems, the DEC 8400 and the Cray T3D. The performance measurements with four processors gives a good impression of the impact of local and remote memory systems performance on the overall performance of the 2D-FFT and reveals at differences between the full Fx-program and the C-code fragment.

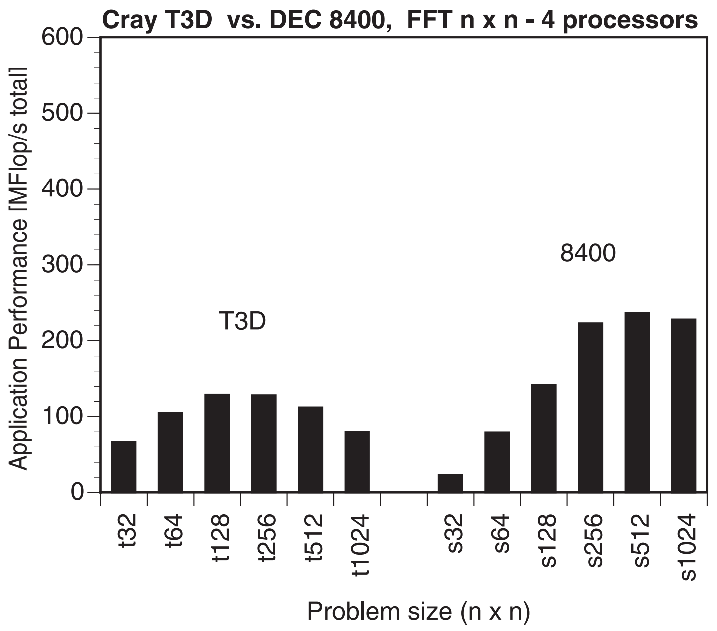


Figure 9.30: Total application performance of the 2D-FFT benchmark on 4 processors of a Cray T3D (left) and a DEC 8400 (right). Benchmark written in Fx Fortran.

The performance picture of the hand-coded C version in Figure 9.31 confirms my earlier observations with the compiler generated Fx Fortran version. I are extremely consistent between C and Fx Fortran versions on the Cray T3D and the compiler generated code for a 2D-FFT performs a well as the hand-optimized, fine tuned C code. On the DEC 8400 there is a slightly larger variability of overall application performance on the DEC 8400 and for certain problem sizes the C code performs slightly better than the compiler generated code. I expect the difference to further diminish as my back-end for the DEC 8400 becomes more mature.

For a complete comparison between sustained performance in a compiled parallel Fx-code and an experimental C-code fragment I break the performance number down into local computation performance (Figure 9.32) and communication performance (Figure 9.33).

The actual decrease in communication performance for small problem sizes (32×32 and 64×64) is

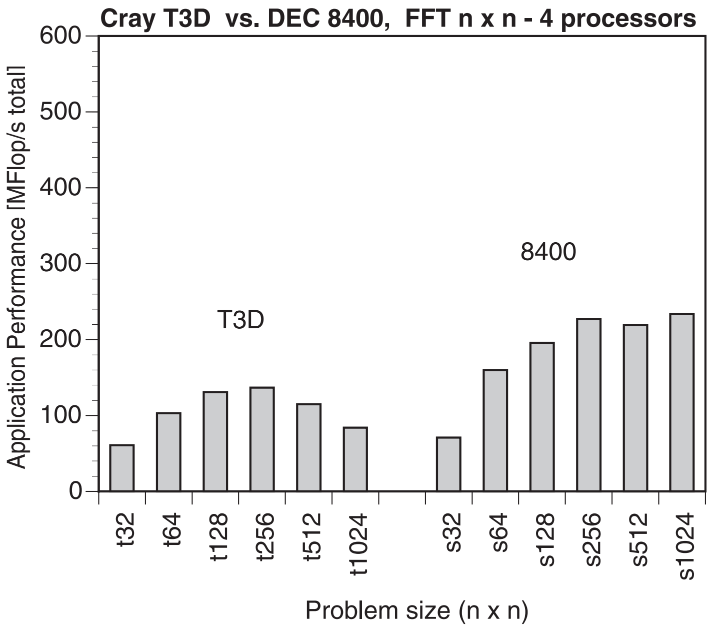


Figure 9.31: Total application performance of the 2D-FFT benchmark on 4 processors of a Cray T3D (left) and a DEC 8400 (right). Benchmark written in C.

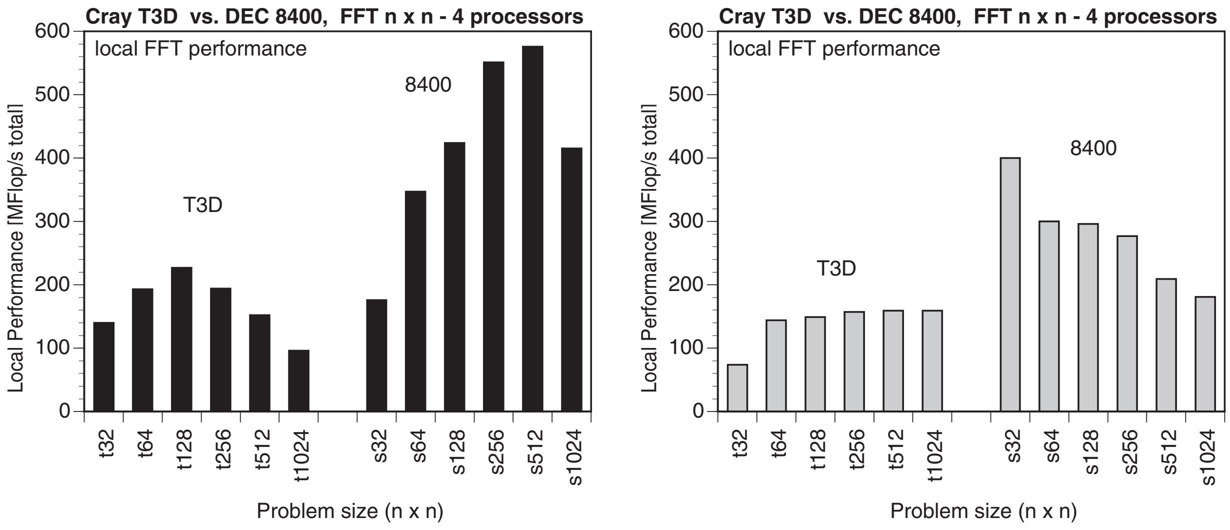
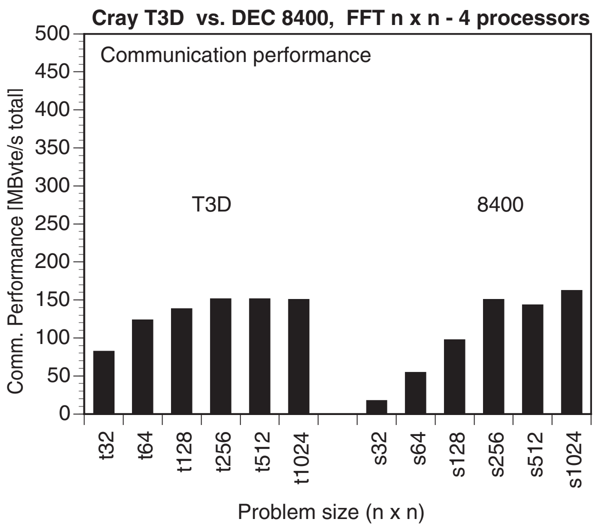


Figure 9.32: Local computation performance of the 2D-FFT benchmark on 4 processors of a Cray T3D and a DEC 8400. Compiler generated benchmark written in Fx Fortran (left) and hand-coded implementation in C (right).

due to my comparison of compiler generated code with some overheads specific to the CATACOMB back-end. CATACOMB provides a general way of generating communication code for all array assignment statements and array distributions, not just for 2D-transposes of block distributed data.



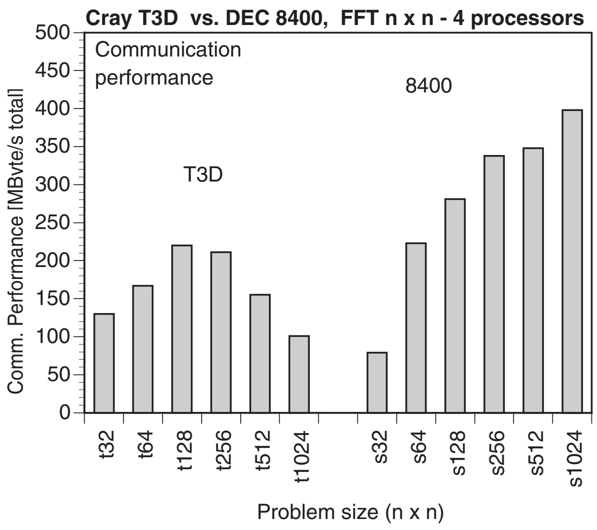


Figure 9.33: Communication performance in the transposes of the 2D-FFT benchmark on 4 processors of a Cray T3D and a DEC 8400. Compiler generated benchmark written in Fx Fortran (left) and hand-coded implementation in C (right).

Chapter 10

Conclusions

An investigation of performance problems with conventional message passing systems points at the synchronization semantics of the traditional message model as probable cause for common inefficiencies in their implementations. The rendez vous model permits direct data transfer without buffering but data transfers are overly synchronized therefore scheduling general communication patterns becomes difficult. The postal model relies on synchronization semantics that can only be maintained by buffering. Based on those observations I demand that synchronization and data transfers must be decoupled to achieve high communication performance and propose the *deposit model* of message passing, a new synchronization model. Along with the deposit model I specify the property of well synchronized programs and show how a parallelizing compiler can produce code for this model. The deposit model establishes a framework for reasoning about control transfers and for a series of optimization to achieve faster control transfers in compiled parallel codes without compromising proper consistency of data transfers and without introducing buffering.

In message passing communication all data transfers originate in memory of the source processor and end up in memory of the destination processor. A parallelizing compiler can choose among different mechanism to move data and must select the optimal mechanism for communication given the access pattern and the memory system of a parallel target machine. Traditional memory models and trace based tools for modeling memory system performance are found to be inadequate and a new approach is needed for reasoning architectural support and for selecting communication code in parallel compilers. I introduce a the *copy transfer model*, a simple model for modeling communication related memory accesses based on two observations (1) the performance of communication related memory operations depends strongly on spatial locality (contiguous, strided or indexed access patterns) (2) communication work is nearly free on temporal locality (the size of working sets is always too large for caches). The copy model proves itself as an indispensable tool for memory system characterization in general and for the memory system optimizations specific to my direct deposit message passing system.

The idea of decoupling synchronization and data transfer in message passing is crucial for all performance optimizations described in this thesis, because tight coupling the two function in message passing implies buffering and extra copies. The main target of our Fx compiler are supercomputers with dedicated processors and strong communication systems, because I believe commodity systems will show similar performance and characteristics in the near future. On supercomputers global synchronization operations (e.g., hardware barriers) are significantly faster than control messages and are the mechanism of choice to achieve synchronization and data consistency within compiled data parallel programs. Furthermore the raw communication speeds of the network interconnect often matches or surpasses the speed of the internal memory bus. Therefore the memory system at the end-points becomes the bottleneck for data transfers, and the most important optimization for speed is to eliminate all extra copies of an end-to-end transfer. This can be done by using built in hardware (a deposit engine or a communication co-processor) for chained data transfers. Chained data transfer extend the concept of zero copy transfers from contiguous buffers to arbitrary data types with strides and indexed data elements of the distributed data structures commonly used in data parallel programs.

The investigation of architectural support in present machines models and measures the memory system performance of the four private memory systems, the Intel iWarp, Intel Paragon, Cray T3D, Cray T3E and of a symmetric shared memory multiprocessor, the DEC 8400 for a comparison. Considering the different generation of systems the distribute memory systems of the Cray systems finish well ahead of the symmetric, coherent shared memory systems in the bus based DEC, when it comes to raw copy transfer performance between processors. The DEC 8400 appeared about 18 month later on the market and is based on newer DEC Alpha microprocessor that the Cray T3D, but still it moves contiguous, strided and indexed data between processors only about at the speed of the T3D. The Cray T3E with a 1995 microprocessor is still faster than most machines that were introduced before the year 2000.

The direct deposit message passing system architecture incorporates decoupled synchronization and chained data transfers. In our performance evaluation it is compared to a good, vendor supplied postal message passing library using the PVM programming interface. I quantify the benefits of the improved software architecture running three application kernels with different access patterns: SOR (contiguous), 2D-FFT (strided) and FEM (indexed). On a machine with fast communication system and good hardware support, the new communication architecture gains up to a factor of three over a highly optimized vendor library.

Chapter 11

Acknowledgements

I would like to thank the faculty of my programm committee for their patience and their continued support during the long systems building effort that lead to this thesis. Systems work would be impossible without the contributions of my fellow graduate students who worked on the iWarp, the Nectar and the Fx projects. They were instrumental to the success of the thesis. I also need to acknowledge the efforts of our staff, in particular of our project staff that spent several person-months on my projects with a high dedication and also to the general computing staff that did not get tired of my demanding requests for help and better service. As this thesis involves experimental work on five different parallel machines, a large number of engineers were involved in this work and helped to make it happen. Among the engineers of Intel Inc., Compaq Digital and Cray Research. Many thanks to the Pittsburgh Supercomputer Center that did not only give us access to the machines but also helped with technical information on their brand new Cray T3D and Cray T3E at the time.

Bibliography

[1] D. Adams. CRAY T3D System architecture overview. Technical report, Cray Research Inc., September 1993.

[2] S. Adve and M. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[3] C. Amza, A. Cox, S. Dwarkadas, C. Hyams, Z. Li, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb 1996.

[4] C. Anderson and A.R. Karlin. Two adaptive hybrid cache coherency protocols. In *Proc. of 2nd International Symposium on High-Performance Computer Architecture (HPCA2)*, pages 303–13, San Jose, CA, USA, Feb 1996. IEEE.

[5] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steve Steinberg, and K. Yelick. Empirical evaluation of the cray-t3d. In *Proc. 22nd Intl. Symposium on Computer Architecture*, pages 320–331, Santa Marguerita di Ligure, June 1995. ACM.

[6] H. Bao, J. Bielak, O. Ghattas, D. O'Hallaron, L. Kallivokas, J. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. In *Supercomputing '96*, Pittsburgh, PA, Nov 1996.

[7] A. Bar-Noy and S. Kipnis. Designing Broadcating Algorithms in the Postal Model for Message-Passing Systems. In *Symp. on Parallel Algorithms and Architectures*, pages 13–22, San Diego, June 1992. ACM.

[8] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: metacomputing on the web. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 181–188, Raleigh, NC, USA, 1996.

[9] R. Barriuso and Knies A. SHMEM user's guide for C. Technical report, Cray Research Inc., June 20 1994. Revision 2.1.

[10] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Procceedings of COMPCON*, 1993.

[11] G. Blelloch and J. Sipelstein. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, Apr 1991.

[12] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leisserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. 5th ACM Symp. on Principles and Practice of Parallel Prog. (PPoPP)*, pages 207–216, Santa Barbara, July 1995. ACM.

[13] M. Blumrich, K. Li, K. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. In *Proc. 21th Intl. Symp. on Computer Architecture*, pages 142–153. ACM, May 1994.

[14] Nanette J. Boden, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet - A gigabit per second local area network. In *IEEE-Micro*, volume 15(1), pages 29–36, February 1995.

[15] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iwarp. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 70–81. ACM, May 1990. A revised version has appeared as technical report CMU-CS-90-197.

[16] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proc. 4th ACM Symp. on Principles and Practice of Parallel Prog. (PPoPP)*, pages 149–158, May 1993.

[17] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Automatic array alignment in data-parallel programs. In *Conf. Record of the 20th Annual ACM Symp. on Principles of Prog. Lang.*, pages 16–28. ACM, January 1993.

[18] B. Chen, A. Borg, and N. Jouppi. A simulation based study of TLB performance. In *19th Annual International Symposium on Computer Architecture*, pages 114–23, Gold Coast, Queensland., Australia, May 1992. ACM.

[19] F. Chism. Communication latency and bandwidth on the cray research t3e. In *Proc. 10th Intl. Parallel Processing Symposium*, pages Slides, Vendor Presentation, Honolulu, HI, April 1996. IEEE.

[20] Intel Corp. *Paragon X/PS Product Overview*. Intel Corp., March 1991.

[21] Parasoft Corporation. *Express User's manual*. Parasoft Corp., 1990.

[22] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*. Thinking Machines Corporation, 1991.

[23] Cray Research Inc. *CRAY T3D Applications Programming Course*, Nov 1993. TR-T3DAPPL.

[24] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, and T. Subramonian, Von Eicker. Logp: Towards a realistic model of parallel computation. Technical Note, 1993. Univ. of California; expanded version of paper in 4th Symp. on PPoPP.

[25] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture – a Hardware/Software Approach*. Morgan Kaufmann Publischers, Inc., San Francisco, CA, USA, 1999.

[26] Z. Cvetanovic and D. Bhandarkar. Performance characterization of the alpha 21164 microprocessor using tp and spec workloads. In *Proc. 2nd High Performance Computer Architecture Conference*, pages 270–279, San Jose, Jan 1996. IEEE.

[27] R. Cypher and E. Leu. The semantics of blocking and nonblocking send and receive. In *Proc. of 8th International Parallel Processing Symposium*, pages 729–35, Cancun, Mexico, April 1994. IEEE.

[28] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, C-36(5):547–553, May 1987.

[29] Digital Equipment Corp., Burlington MA. *Alpha Architecture Reference Manual*, 1992. Sites, Richard (Editor), EY-L520E-DP.

[30] Digital Equipment Corp., Maynard MA. *Alpha 21164 Microprocessor, Hardware Reference Manual*, 1995. EC-QAEQB-TE.

[31] Digital Equipment Corp., Maynard MA. *AlphaServer 8200 and Alpha Server 8400, Technical Summary*, 1995. BC-N446-10.

[32] P. Dinda and D. O'Hallaron. Fast message assembly using compact address relations. In *Proceedings of the Intl. Conf. on Measurement and Modeling of Computer Systems, ACM SIGMETRICS*, pages 213–226, Philadelphia, PA, May 1996. ACM.

[33] C. Dubnicki, L Iftode, E. W. Felten, and K. Li. Software support for virtual memory-mapped communication. In *10th International Parallel Processing Symposium on*, Cancun, Mexico, April 1996. IEEE.

[34] B. Falsafi and D. Wood. Scheduling communication on an smp node parallel machine. In *Proc. of 3nd International Symposium on High-Performance Computer Architecture (HPCA3)*, pages 128–138, San Antonio, Tx, USA, Feb 1997. IEEE.

[35] A. Feldmann, O. Ghattas, J. R. Gilbert, G. L. Miller, D. R. O'Hallaron, J. R. Shewchuk, and S. Teng. Automated parallel solution of unstructured PDE problems. Technical Report working paper, School of Computer Science, Carnegie Mellon University, September 1996. http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/pde.bw.ps.

[36] A. Feldmann, T. Stricker, and T. Warfel. Supporting sets of arbitrary connections on iwarp through communication context switches. In *Proc. SPAA*, pages 203–212. ACM, June 1993.

[37] E. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. PhD thesis, Univ. of Washington, Seattle, WA, 1993.

[38] H. W. Felten. Generalized signals: an interrupt-based communication system for hypercubes. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 563–568, New York, NY, USA, 1988. ACM.

[39] D. Fenwick, D. Foley, W. Gist, S. VanDoren, and D. Wissell. The alphaserver 8000 series: High-end server platform development. *Digital Technical Journal Vol.*, 7(43), Spring 1995.

[40] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Intl. Symposium on Theory of Computing*, pages 114–118. ACM, 1978.

[41] High Performance Fortran Forum. High Performance Fortran language specification version 1.0 draft, January 1993.

[42] The MPI Forum. MPI: A Message Passing Interface. In *Proc. Supercomputing '93*, pages 878 – 883, Oregon, November 1993. ACM/IEEE.

[43] S. Frank, H. Burkhardt, and J. Rothnie. The KSR 1: bridging the gap between shared memory and mpps. In *CCOMPCON Spring '93. Digest of Papers.*, pages 285–94, San Francisco, CA, Spring 1993. IEEE Comput. Soc. Press.

[44] W.K. Giloi, U. Bruening, and W. Schroeder-Preikschat. Manna: prototype of a distributed memory architecture with maximized sustained performance. In *Proceedings of 4th Euromicro Workshop on Parallel and Distributed Processing*, pages 297–304, Braga, Portugal, Jan 1996. IEEE.

[45] T. Gross. Communication in iwarp systems. In *Proc. Supercomputing '89*, pages 436–445. ACM/IEEE, November 1989.

[46] T. Gross, A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker. Communication styles for parallel systems. *IEEE Computer*, 27(12):34–44, Dec 1994.

[47] T. Gross and D. O'Hallaron. *iWarp: Anatomy of a Parallel System (working title).* (currently under review), 1996.

[48] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a high performance fortran framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.

[49] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of message passing and shared memory in the stanford flash multiprocessor. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 38–50, San Jose, CA, USA, Oct 1994. ACM.

[50] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J.P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The performance impact of flexibility in the stanford flash multiprocessor. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 274–84, San Jose, CA, USA, Oct 1994. ACM.

[51] J. L Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach (2nd Edition).* Morgan Kaufman, 1995.

[52] S. Hinrichs. *Programmed Communcation Service Tool Chain User's Guide*. Carnegie Mellon University, release 2.8 edition, 1991. Now part of Intel RTS 3.0.

[53] S. Hinrichs. *Compiler-Directed Architecture-Dependent Communication Optimization*. PhD thesis, Carnegie Mellon, School of Computer Science, June 1995. CMU-CS-95-155.

[54] S. Hinrichs. Simplifying Connection-Based Communication. *IEEE Parallel and Distributed Technology*, 3(1):25–36, Spring 1995.

[55] S. Hinrichs, C. Kosak, D. O'Hallaron, T. Stricker, and R. Take. An Architecture for Optimal All-to-All Personalized Communication. In *ACM Symp. on Parallel Algorithms and Architectures*, pages 310–319, Cape May, New Jersey, June 1994. A revised version is available as Tech. Report CMU-CS-94-140.

[56] S. Hinrichs, C. Kosak, D. O'Hallaron, T. Stricker, and R. Take. An Architecture for Optimal All-to-All Personalized Communication. Technical Report CMU-CS-94-140, Carnegie Mellon University, School of Computer Science, 1994.

[57] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, August 1978.

[58] M. Homewood, D. May, D. Shepherd, and R. Shepherd. The ims t800 transputer. *IEEE Micro*, 7(5):10–26, October 1987.

[59] L Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *Proc. of 2nd International Symposium on High-Performance Computer Architecture (HPCA2)*, pages 303–13, San Jose, CA, USA, Feb 1996. IEEE.

[60] Cray Research Inc. *PVM and HeNCE Programmer's Manual*. Mendota Heights, MN, manual v4 for release 1.1 edition, 1994. SR-2501-4.

[61] N. Islam. *Customized Message Passing and Scheduling for Parallel and Distributed Applications*. PhD thesis, Univ. of Illinois, Urbana-Champain, 1994.

[62] V. Karamcheti and A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and Cray T3D. In *Proc. 22nd Intl. Symposium on Computer Architecture*, pages 298–307, Santa Marguerita di Ligure, June 1995. ACM.

[63] V. Karamcheti and A.A. Chien. Software overhead in messaging layers: where does the time go? In *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 51–60, San Jose, October 1994. ACM.

[64] H. Karl. Bridging the gap between distributed shared memory and message passing. *Concurrency: Practice and Experience*, 10(11–13):887–900, Sep 1998.

[65] K. Knobe. Issues in generating code for distributed memory architectures. Dagstuhl 91 Workshop on Compiler Construction, May 1991.

[66] K. Knobe, J. Lukas, and G. Steele Jr. Data optimization: Allocation of arrays to reduce communication on simd machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, Feb 1990.

[67] C. Koelbel, D. Loveman, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[68] S. Koenig, R. McDaniel, and Kim Wagner. The workings of the pathway scheduler. Private communications - student project, 1991.

[69] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *Proc. 21th Intl. Symp. on Computer Architecture*, pages 302–313. ACM, May 1994.

[70] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *ACM*, 21(7), July 1978.

[71] C. Leiserson, A. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. St.Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Symp. on Parallel Algorithms and Architectures*, pages 272–285, San Diego, June 1992. ACM.

[72] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of 1988 International Conference on Parallel Processing*, pages 94 – 101, 1988.

[73] P.N. Loewenstein and D.L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. *Formal Methods in System Design*, 1(4):355–83, Dec 1992.

[74] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proc. Intel Supercomputer Users' Group. 1994 Annual North America Users' Conf.*, pages 245–251, June 1994. ftp://ftp.cs.sandia.gov/pub/sunmos/papers/published/ISUG94-1.ps.

[75] R. Martin. Hpam: an active message layer for a network of hp workstations. In *Proc. of Hot Interconnects Symposium II*, Palo Alto, August 1994. IEEE/TCMM.

[76] G. McRae, W. Goodin, and J. Seinfeld. Development of a second-generation mathematical model for urban air pollution - model formulation. *Atmospheric Environment*, 16(4):679–696, 1982.

[77] T. Mummert, C. Kosak, P. Steenkiste, and A. Fisher. Fine grain parallel communication on general purpose LANs. In *Proc. Intl. Conf. on Supercomputing*, page 341-349, Philadelphia, May 1996. ACM.

[78] nCUBE Corp. *nCUBE 2: Technical Overview*. nCUBE Corporation, Foster City, Ca., 1992.

[79] B. Nelson. *Remote procedure call*. PhD thesis, Carnegie-Mellon University - Disertations, 1981.

[80] R. Numrich, P. Springer, and J. Peterson. Measurement of communication rates on the Cray T3D interprocessor network. In *Proc. HPCN Europe '94, Vol. II*, pages 150–157, Munich, April 1994. Springer Verlag. Lecture Notes in Computer Science, Vol. 797.

[81] R.W. Numrich. F - - : a parallel extension to Cray Fortran. *Scientific-Programming*, 6(3):275–284, 199.

[82] W. Oed. The Cray Research massively parallel processor system Cray T3D, 1993. Available from via ftp from cray.com.

[83] D. O'Hallaron and J. Shewchuk. Properties of a family of parallel finite element simulations. Technical Report CMU-CS-96-141, School of Computer Science, Carnegie Mellon University, September 1996.

[84] S. Pakin, V. Karamcheti, and A. Chien. Fast messages: efficient, portable communication for workstation clusters and mpps. *IEEE-Concurrency*, 5(2):60–72, Apr-Jun 1997.

[85] P. Pierce and G. Regnier. The Paragon implementation of the NX message passing interface. In *Proc. of Scalable High Performance Computing Conference (SHPCC)*, pages 184–190, Knoxville, TN, USA, May 1994. IEEE.

[86] D. Pountain and D. May. *A Tutorial Introduction to Occam Programming*. BSP Professional Books, Oxford, GB, 1987.

[87] M.J. Quinn and P.J. Hatcher. On the utility of communication-computation overlap in data- parallel programs. *Journal of Parallel Distributed Computing*, 33(22):197–204, Spring 1996.

[88] W. J. Savitch and M. Stimson. Time bounded random access machines with parallel processing. *Journal of the ACM*, 26:103–118, Spring 1979.

[89] E. J. Schwabe, G. E. Blelloch, A. Feldmann, O. Ghattas, J. R. Gilbert, G. L. Miller, D. R. O'Hallaron, J. R. Shewchuk, and S. Teng. A separator-based framework for automated partitioning and mapping of parallel algorithms for numerical solution of pdes. In *Proc. 1992 DAGS/PC Symp.*, pages 48–62, June 1992. Revised version accepted for Comm. ACM.

[90] S. Scott. Synchronization and communication in the Cray T3E multiprocessor. In *Proc. 7th. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Boston, MA, Oct 1996. ACM.

[91] C. Seitz. The cosmic cube (concurrent computing). *Communications of the ACM*, 28(1):22–33, Jan 1985.

[92] M. Snir. Scalable parallel computing - the ibm 9076 scalable powerparallel-1. In *ACM Symp. on Parallel Algorithms and Architectures*, page 42. ACM, June 1993.

[93] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, Orlando, FL, USA, June 1994. ACM.

[94] J. Stichnoth. *Generating Code for High-Level Operations through Code Composition*. PhD thesis, Carnegie Mellon School of Computer Science, September 1997.

[95] J. Stichnoth and T. Gross. A communication backend for parallel language compilers. In *Proc. 8th Intl. Workshop Languages and Compilers for Parallel Computing (LCPC '95)*, pages 224–238, Columbus, OH, Aug 1995. Springer Verlag.

[96] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, 1994.

[97] T. Stricker. Message routing on irregular 2d-meshes and tori. In *Proc. 6th Distributed Memory Computing Conf.*, pages 170–177, Portland, OR, April 1991. Also appeared as Technical Report CMU-CS-91-109, Carnegie Mellon School of Computer Science.

[98] T. Stricker and T. Gross. Optimizing memory system performance for communication in parallel computers. In *Proc. 22nd Intl. Symposium on Computer Architecture*, pages 308–319, Santa Marguerita di Ligure, June 1995. ACM.

[99] T. Stricker and J. Hardwick. From AAPC algorithms to high performance permutation routing and sorting. In *Proc. of Symposium on Parallel Algorithms and Architectures (SPAA)*, pages ?–?, Padua, Veneto, Italia, June 1996. ACM.

[100] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proc. Intl. Conf. on Supercomputing*, pages 1–10, Barcelona, July 1995. ACM.

[101] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proc. 4th ACM Symp. on Principles and Practice of Parallel Prog. (PPoPP)*, pages 13–22, May 1993.

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

[103] D. Tardidi. Multi lane routing in iWarp. Personal communications - student project, 1993.

[104] C. Thekkath, H. Levy, and E. Lazowska. Separating data and control transfer in distributed operating. In *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 1–12, San Jose, October 1994. ACM.

[105] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Trap-driven simulation with TAPEWORM II. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 132–44, San Jose, CA, USA, Oct 1994. ACM.

[106] T. von Eicken, A. Basu, and V Buch. Low latency communication over ATM networks using active messages. *IEEE Micro*, 15(1):46–53, Feb 1995.

[107] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proc. of 15th Symposium on Operating Systems Principles (SOSP-15)*, pages ?–?, Cooper Mountain, CO, USA, Dec 1995. ACM.

[108] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Intl. Conf. on Computer Architecture*, pages 256–266, May 1992.

[109] D.A. Wallach, W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, and W.E. Weihl. Optimistic active messages: a mechanism for scheduling communication with computation. In *Proc. Symp. on Principles and Practice of Parallel Programming, PPoPP*, pages pp. 217–26, Santa Barbara, CA, USA, July 1995. ACM.

[110] H. Wasserman. Benchmark tests on the digital equipment corporation alpha axp 21164-based alphaserver 8400. In *Proc. 1996 International Conference on Supercomputing*, pages 333–340, Philadelphia, May 1996. ACM.

[111] J. Webb. Implementation and performance of fast parallel multi-baseline stereo vision. In *Proc. 1993 Computer Architectures for Machine Perception*, pages 232–240, New Orleans, Dec 1993. IEEE Comp. Society.

[112] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon, May 1991.

[113] X. Zhang and X. Qin. Performance prediction and evaluation of parallel processing on a NUMA multiprocessor. *IEEE Transactions on Software Engineering*, 17(10):1059–68, Oct 1991.