Architectural Support for Compiler-Generated Data-Parallel Programs

by

Alexander C. Klaiber

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1994

Approved by _____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree _____

Date _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature_____

Date_____

University of Washington

Abstract

Architectural Support for Compiler-Generated Data-Parallel Programs

by Alexander C. Klaiber

Chairperson of Supervisory Committee: Professor Henry M. Levy
Department of Computer Science
and Engineering

To fully realize the advantages of parallel processing demands the design of efficient communication mechanisms. Existing communication architectures span a spectrum ranging from message passing to remote-memory access, shared memory and cache-only architectures. These communication architectures are often used *(and designed to be used)* directly by the programmer. However, in the future we can expect more programs to be written in high-level parallel languages and compiled to the specific parallel target; the compiler will hide the details of the underlying hardware from the programmer. The communication architecture should then be designed with the compiler, not the programmer, in mind.

The goal of our work is to improve communication performance for programs that are written in a high-level parallel language and then compiled to a specific communication architecture. To make this task manageable, we focus on the class of *data-parallel languages* and we pick C* as one representative for our experiments.

We evaluate three competing communication architectures — message-passing, remote-memory access and cache-coherent shared-memory — for a set of benchmarks written in C* and compiled to the respective architecture. We show that the message-passing model has several inherent advantages for these benchmarks, resulting in less interconnect traffic and less time spent waiting for messages to traverse the interconnect.

On the other hand, the message-passing architecture requires the CPU to perform significantly more work per message than the other architectures. This communication overhead destroys much of the message passing model's advantage.

We propose a *language-oriented* communication architecture that retains the advantages

of the message-passing model, yet (in cooperation with the compiler) significantly reduces the communication overhead. To do so, we first identify a small set of low-level communication and synchronization primitives that are well matched to the needs of C*. We then design a network interface that is tuned to these primitives and describe the C* compilation for this base; our network interface includes hardware for remote read/write requests plus counter-based synchronization support. We simulate and measure our compiled C* benchmarks on a traditional message-passing interface as well as our language-oriented design; our measurements demonstrate that our design is effective at reducing communication-related CPU overhead.

Table of Contents

List of Figures

ACKNOWLEDGMENTS

I'd like to thank all the people who have made this possible. My parents for their unwavering support. Beth for keeping me sane. My advisor Hank Levy for teaching me how to do research and, more importantly, how to write. All my fellow grad students for many hours of productive discussions, especially Craig Anderson, Robert Bedichek, Edward Felten, Dave "Pardo" Keppel, Brian Lockyear, Neil McKenzie and Dylan McNamee. Extra thanks to Dylan for his *thorough* reading of the first draft of this dissertation.

Thanks to Jamie Frankel for getting me interested in data-parallel languages in the first place. Phil Hatcher at UNH supplied me with an early version of his compiler for the "new" C* and most of the benchmarks.

Finally, thanks to Wassif at the Shalimar for keeping me well fed!

To my parents.

Chapter 1

INTRODUCTION

Massively parallel computers are an attractive tool for solving many computationally intensive problems. Unlike traditional supercomputers which generally use expensive custom-designed processors, parallel computers can be built from off-the-shelf microprocessors and achieve supercomputer class performance through parallelism. Because of their design, parallel computers gain a significant market advantage by tracking the steady performance improvement of mass-produced commodity parts.

Unfortunately, the cost of *communication* may limit the performance of parallel computers. To fully realize the advantages of parallel processing, we need to design efficient communication mechanisms. Existing *communication architectures* span a spectrum ranging from message passing [Arlauskas 88, Intel 91a, Dally 90, TMC 91b] to remote-memory access [Crowther et al. 85, Cray 93], shared memory [Sequent 87, Lenoski et al. 92, Agarwal et al. 91] and cache-only architectures [Hagersten 92a, KSR 92]. These communication architectures are often used directly by the programmer — a fact that has influenced their design, much as assembly language programming has influenced the design of CISC instruction sets. For example, one commonly cited argument in favor of shared-memory machines is that they are easier to use than message-passing machines — a statement that clearly reveals a design bias towards simplifying the communication architecture for the benefit of the programmer.

However, in the future we can expect more programs to be written in high-level parallel languages and compiled to the specific parallel target; the compiler will hide the details of the underlying communication architecture from the programmer. Hence, the programmer's convenience is no longer a major concern in the design of the communication architecture, since the programming language already provides a convenient programming model. Instead, performance becomes a driving concern, and the communication architecture must provide interfaces that best suit the needs of the compiler. This approach is similar to the RISC philosophy in processor design.

1.1 Goal of this Dissertation

The goal of our work is to improve communication performance for programs that are written in a high-level parallel language and then compiled to a specific communication architecture.

To make this task manageable, we focus on the class of *data-parallel languages*, and we pick the C* language as one representative for our experiments. The data-parallel model is an important one; a study by Fox [Fox 88] has shown that a majority of existing scientific applications fit that model well. As a framework for our architectural studies, we concentrate on MIMD parallel computers and three competing communication architectures — message-passing, remote-memory access and cache-coherent shared-memory.

The core of this dissertation consists of two parts. In the first part (Chapter 5), we evaluate the three communication architectures in order to gain better insight into their relative strengths, as well as the compiler's demands on the communication architecture. Comparing such widely differing architectures has been difficult in the past, for two reasons. First, applications had to be hand-crafted for each architecture, often resulting in radically different sources for comparison. Second, a host of implementation details (such as processor speed, cache organization and size, and network bandwidth available) can easily obscure any *architecture-inherent* characteristics; given different execution times on different machine configurations, it becomes nearly impossible to attribute the performance differences to any specific source(s).

We avoid these problems by using the following approach. We use a single suite of C* source programs, compile each program with a C* compiler, and simulate its execution on the alternative communication architectures. This ensures that the different architectures execute the same source program. Further, our objective is to examine underlying, *implementation-independent* costs inherent in each alternative. To this end, we abstract many implementation details and focus on metrics that are not affected by, say, processor or network speed. For example, we measure the number of messages sent, the total interconnect control and data traffic, and the number of round-trip communication latencies incurred. We deliberately reject overall execution time as a metric, since it cannot yield the same kind of fundamental insight as our implementation-independent metrics. While our results do not directly indicate which architecture is "best," they do show the relative communication work required to execute our data parallel programs on the different architectures. Specifically, we will see that the message-passing model has some inherent

advantages for these benchmarks, resulting in less interconnect traffic and less time spent for messages traversing the interconnect.

In the second part of the dissertation (Chapter 6), we focus specifically on distributed-memory architectures, and we examine one metric that we have ignored in Chapter 5, namely the time to send a message, or the *communication overhead*. We note that in practice, the message-passing communication model incurs significant per-message overhead and thus is unable to fully exploit the advantages we have identified before.

A good communication architecture would combine the advantages of message-passing with the low per-message overhead of shared-memory. As one possible solution, we propose a *language-oriented* design that retains the advantages of the message-passing model, yet (in cooperation with the compiler) significantly reduces the per-message overhead. To do so, we first identify a small set of low-level communication and synchronization primitives that are well matched to the needs of C* (and, presumably, other data-parallel languages as well). We then design a network interface that is tuned to these primitives and describe the C* compilation for this base; our network interface includes hardware for remote read/write requests, plus counter-based synchronization support.

Finally, to evaluate the effectiveness of this approach, we simulate and measure our compiled C* benchmarks on a traditional message-passing interface as well as our language-oriented design. These measurements demonstrate that our design is effective at reducing communication-related CPU overhead; for example, traffic between the CPU and network interface is reduced by 50 to 90 percent on these benchmarks.

1.2 Related Work

Much research has been done in the past to improve performance of both shared-memory and message-passing architectures. For message-passing systems, researchers have largely focused on reducing the high per-message overhead typically found in message-passing systems. For example, active messages [von Eicken et al. 92] are a low-level transport mechanism that achieves low latency by efficiently dispatching to a message handler on the receiving node. Felten [Felten 93a] proposes using a *protocol compiler* to custom-generate message-passing protocols for a given program and thus reduce protocol overhead.

The above two approaches rely entirely on software techniques; however, hardware approaches have been suggested as well. For example, the Shrimp architecture [Blumrich et al. 94] implements a low-overhead data transport mechanism which for selected

memory pages automatically forwards a processor's `store` operations to other nodes. The resulting system shares some of the characteristics of remote-memory and shared-memory architectures.

There also is a large body of work aimed at improving the performance of cache-coherent shared-memory architectures. For example, researchers have studied adaptive or user/compiler selectable cache coherence mechanisms that use different coherency protocols for different sharing patterns [Carter et al. 91, Bennett et al. 92, Stenstrom et al. 93]. Some machines like the KSR-1 [KSR 92] provide processor instructions to prefetch or poststore data, or load data in a state that facilitates future writes. Most of these techniques try to improve performance by giving the application more explicit control over how and when data is moved between processing nodes. As a result, shared-memory systems take on some of the characteristics of message-passing systems (where data movement is entirely under explicit application control).

A related approach [Frank & Vernon 93] integrates message passing and shared memory by introducing a new cache line state, *possibly-stale*, into a conventional cache coherence protocol. The proposed architecture lets user programs move data between nodes without the overhead of cache coherence operations. At the same time, caches are kept coherent to provide a traditional shared memory model.

Relaxed memory consistency models [Gharachorloo et al. 90, Adve & Hill 90, Hutto & Ahamad 90] attempt to improve shared memory performance by allowing temporary inconsistencies among multiple copies of the same data. This is similar in nature to what happens in message-passing systems where copies of remote data are created under the control of the application program.

Yet another approach, taken by the Alewife machine, is to offer both a shared-memory system and message-passing primitives. In [Kranz et al. 93], the authors identify several scenarios where a compiler or programmer could implement operations more cheaply through message passing than through shared memory.

Finally, the FLASH [Kuskin et al. 94] and Typhoon [Reinhardt et al. 94] shared-memory architectures include fully programmable network interfaces. In principle, this would allow coherence protocols to be tailored to specific applications; it is even conceivable to turn these machines into NUMA or message-passing machines simply by reprogramming the network interfaces.

The current trends in research discussed above indicate that the different communi-

cation architectures are starting to converge. Shared-memory architectures are acquiring message-passing like features such as better user-level control over data movement, whereas message-passing architectures are striving for data transport mechanisms with low overheads comparable to shared-memory and remote-memory architectures.

In our work, we compare these different architectures at a high level of abstraction; our goal is to clarify the tradeoffs between the architectures and point out desirable features of each architecture. Our results should be helpful in future research on communication architectures that unify the advantages of message-passing and shared-memory.

1.3 Contributions of this Dissertation

In this dissertation, we make the following contributions:

- We recognize that most existing communication architectures have been designed to be used directly by the programmer. However, in the future, more programs will be written in high-level parallel languages and compiled to a specific communication architecture. The compiler hides the details of the communication architecture from the programmer. We show that this change in programming style both requires and enables changes to the communication architecture in order to improve performance.

- We evaluate the strengths and weaknesses of three competing communication architectures — message-passing, remote-memory access, and cache-coherent shared-memory — for a workload of compiled C* programs. We show that compared to the other architectures, the message-passing model has various advantages. For example, the compiler has better control over data movement and granularity, and the run-time system can combine data transfer and synchronization in a single operation. As a result, the message-passing model requires less interconnect bandwidth and incurs lower communication latencies than the other models. In the process, we also present a strategy for obtaining meaningful comparisons across such widely differing of architectures.

- Noting that the message-passing model incurs significant per-message CPU overhead, we propose a *language-oriented* approach to designing a communication architecture. We first identify a small set of communication primitives that match the needs of the

C* compiler. We then present the design of a network interface for a distributed-memory architecture that is tuned to those communication primitives. Together with a compilation approach normally used on shared-memory machines, we are able to retain the above advantages of the message-passing model while drastically reducing the per-message CPU overhead.

1.4 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we discuss parallel machine architecture, which sets up the hardware framework for our studies. In Chapter 3, we describe the C* language and highlight its close similarity to High Performance Fortran, another data-parallel language. We then outline, in Chapter 4, how our compiler translates C* for different target communication architectures. In Chapter 5, we evaluate the relative strengths and weaknesses of three different communication architectures — message-passing, remote-memory access, and cache-coherent shared memory. Our results show that the message-passing model generates less interconnect traffic, sends fewer messages and incurs less network latency than either of the competing models, primarily because it offers the compiler better control over data movement and granularity of communication. These findings imply that message-passing has some inherent advantages over the other models, at least for the benchmarks studied. In practice, however, this advantage can not currently be realized, due to the extremely high per-message CPU overhad in existing message-passing systems. In Chapter 6, we propose a *language-oriented* approach to designing a communication architecture. Our network interface includes hardware support for a small set of communication primitives that match the needs of the C* compiler. Our simulations show that our design retains the advantages of the message-passing model while drastically reducing the per-message CPU overhad. We summarize our results, present our conclusions and discuss future work in Chapter 7.

Chapter 2

ARCHITECTURAL FRAMEWORK

Figure 2.1 shows an MIMD parallel computer, organized into three major component groups: *interconnection network*, *processing nodes* and *network interfaces* ("NI"). The network interfaces connect the processing nodes to the interconnect. In this chapter, we describe some of the architectural tradeoffs for each of the three components. Many of the design decisions are beyond the scope of this dissertation so we discuss them only briefly in order to provide a framework for our architectural research; we focus primarily on the network interface.

2.1 Processing Nodes

A fundamental decision in designing the processing node is whether to use commodity or custom processors. A custom processor design can improve communication performance; for example, the architect can integrate the network interface more tightly with the CPU [Henry & Joerg 92a], or include special-purpose communication instructions in the CPU. The Kendall Square KSR-1 shared-memory computer [KSR 92] uses both approaches; its processors provide instructions for prefetching or post-storing cache lines, plus a host of instructions that control the memory system, especially the caching strategy. Different variants of the `load` instruction exist that can request a *writable* copy of a cache line (useful if the cache line is written later), or specify that the level-0 processor caches are to be bypassed by the access. The designers have added these instructions to the processor in order to provide the programmer or compiler with better control over data movement, with the ultimate goal of improving performance.

However, many of these functions can also be implemented on systems that use only commodity processors — albeit possibly with slightly lower performance. For example, the operating system can create multiple virtual memory mappings for a given area of physical memory, with each mapping providing different access semantics. Performance may not match that of special-purpose `load` and `store` instructions, since the processor must likely perform more address arithmetic to access the different mappings. Likewise, the network

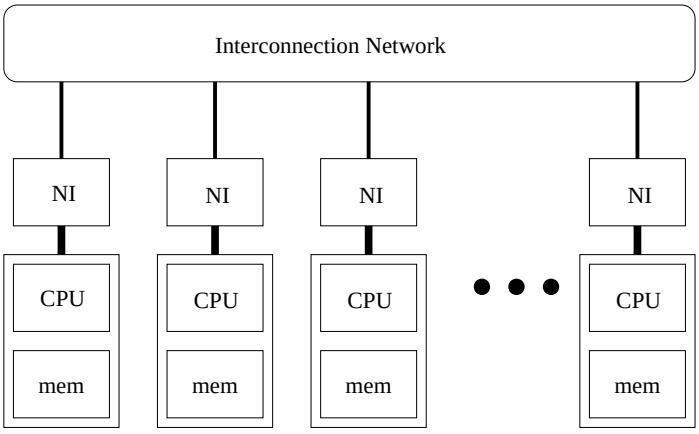


Figure 2.1: A generic parallel machine.

interface can be made accessible as a memory-mapped device, such that the processor can initiate special operations (e.g., prefetching or flushing data) by writing memory-mapped control registers in the network interface. Again, the performance of off-chip logic may be worse than dedicated on-chip hardware, but both designs provide the same functionality.

Using off-the-shelf processors reduces overall design time for the machine and results in faster time-to-market. In fact, most existing parallel computers, such as Intel's series of message-passing machines [Arlauskas 88, Bokhari 90, Intel 91b, Intel 91a], the Thinking Machines CM-5 [TMC 91b], or the Cray T3D [Cray 93] (a NUMA machine), use processing nodes built around a commercial off-the-shelf microprocessor. Instead of requiring special-purpose processor instructions, these machines control communication through hardware external to the processor.

We conclude that the choice between custom or commodity processors is not likely to affect *functionality*, though it could possibly affect the speed at which some communication operations can be executed. In our studies, we focus on the functionality of the communication architecture, and abstract implementation-dependent details such as timing; our results apply equally well to either design alternative.

2.2 Interconnection Network

The *interconnection network* is the physical substrate for moving data between processing nodes. Some key design decisions for the network are its *topology*, *reliability*, *routing strategy* and *packet size* (where applicable). In this section, we discuss some of the tradeoffs and how they interact with the design of network interfaces, described in the next section.

Reliability

Different networks offer different degrees of *reliability*. For example, due to congestion or transmission errors, the network may lose or corrupt data packets, or it may deliver spurious messages; the network may or may not detect and report these conditions. ATM network switches [Minzer 89] are allowed to "drop" packets without notification if network congestion gets too high.

If the network does not deal with such events, the communication endpoints (i.e., the NI or processing node) must execute a protocol that can handle them. Note that this involves work beyond the actual data transfer — extra processing time in the NI or node, plus possibly transmission of additional protocol messages. If the network can lose packets, the sender must be prepared to re-send a given message until it has received an *acknowledgement* that the message has indeed been delivered. Usually, the sender must save a *copy* of each message until it is acknowledged; this in turn may incur overhead for managing the required buffer space. Brustoloni and Bershad [Brustoloni & Bershad 92] have developed an efficient protocol for ATM-based networks that can handle message loss.

For our studies, we assume that the communication primitives used by the programmer or compiler are reliable, i.e., the network neither loses nor corrupts messages, and it does not introduce spurious messages.

Topology

Many different network topologies have been explored in the literature. Some topologies, for example buses and rings, provide inherent broadcast capabilities, whereas others, such as meshes, hypercubes or trees are point-to-point interconnects. Different topologies have different scaling characteristics, and often perform particularly well on some set of communication patterns — e.g., 2-D nearest-neighbor communication maps well onto mesh

interconnects. The topology of the network does not significantly affect the design of the network interface (though the NI designer may wish to include some mechanism to access the interconnect's broadcast capabilities, if applicable.) For this reason, we do not assume any particular network topology in our studies; the results we present will be independent of topology though they may translate into different execution times depending on the topology. We do, however, examine the impact of broadcast capability.

Routing and Ordering

Networks use a *routing* algorithm to direct messages from their source node to their destination. We consider two important classes of routing strategies: *oblivious* and *adaptive* algorithms. Given a source and destination node number for a data packet, an oblivious algorithm will always choose the same path for the packet, whereas an adaptive algorithm may route the packet along any of several paths, depending on network load. Adaptive routing algorithms have the advantage that they may use the interconnect's aggregate bandwidth more efficiently [Ngai & Seitz 89, Snyder 92, Konstantinidou & Snyder 91]. However, they also do not generally guarantee *FIFO* delivery of messages; in other words, if node A sends messages m_1 and m_2 to node B in that order, the messages may arrive in reverse order at node B.

Clearly, this can be a problem for application programs. For example, it is harder to implement efficiently a sequentially consistent shared-memory system if data packets can be delivered out of order [Thapar et al. 93]. Weaker memory consistency models [Hutto & Ahamad 90, Gharachorloo et al. 90] may be able to tolerate out-of-order delivery more easily.

Note that one can implement in-order message delivery on top of a network that delivers messages out of order, for example by adding sequence numbers to packet headers, and by reordering packets according to the sequence numbers at the receiving node. However, this approach requires the receiver to buffer packets that arrive "early" and hence also incurs overhead (processing time and possibly extra protocol messages) for managing the buffer space.

Except where noted otherwise, we assume that the communication primitives used by the programmer or compiler guarantee in-order message delivery.

Packet Size

Modern networks using packet-switching techniques have to split large messages into smaller *packets* that are then routed through the network. The CPU or NI of the sender of a large message must perform *packetization* (i.e., split the message body into multiple small packets), and the receiver must generally reassemble (*unpacketize*) the message from the individual packets.[1] Note that even with FIFO delivery in the network, a node may receive packets from different messages (sent by different nodes) *interleaved* with each other. Some programming models (including "traditional" message-passing) require receipt of a message to be *atomic*. Providing those semantics in the presence of packetization again incurs buffering and protocol overhead. Clearly, the larger the packet size, the less packetization-related overhead is incurred. Also, larger packet sizes amortize any message header overhead over a larger body. On the other hand, smaller packets may be easier to route.

A network's packet size may further be *fixed* or *variable* — in the former case, all packets sent through the network are of one fixed size, even if only part of the packet body carries useful data. In the latter approach, the packet header includes a field indicating the size of the packet body; this may result in more efficient utilization of the network's bandwidth. Overall, there appears to be no clear consensus which of these approaches is better, or what packet sizes are desirable; in fact, the "ideal" packet size depends heavily on the workload [Cypher et al. 93]. In our work, we therefore examine different packet sizes.

Special-Purpose Networks

Some machines, such as the Thinking Machines CM-5 [TMC 91b] or the Cray T3D [Cray 93], use dedicated networks for certain communication operations. For example, the CM-5 includes a *control network* that has been specifically optimized to perform efficient reduction and broadcast operations. Similarly, the T3D has a *synchronization* network that provides very low-latency barrier synchronization operations. In both cases, these special-purpose networks have been included in order to improve the performance of some common communication operations.

---

[1] Note the interaction between packetization and out-of-order delivery: due to the former, a large message must be split into separate packets, and due to the latter, the packets may arrive at the receiver in arbitrary order, thus complicating unpacketization.

However, this can be a rather expensive approach, and a dedicated synchronization network may not always result in as significant a performance improvement as expected. For example, on the Cray T3D, the barrier hardware propagates the signal throughout a 256-node machine in 26 cycles, yet the `barrier` routine takes about 240 cycles because it has to flush the write buffer and wait for asynchronous remote writes to complete. The write messages, of course, propagate through the *data* network which has much higher latency than the dedicated barrier network [Barrusio 94]. This drastically limits the potential performance gains from the dedicated network.

Protection

Parallel machines that support time- or space-sharing among multiple users must address the issue of protecting different jobs from each other. One user's job should not be allowed to send messages to another user's job. Likewise, the network traffic generated by one job should not keep another job from making progress, e.g., by deadlocking the network. Solving this problem may in general require cooperation between the network interface and the network.

The topology of the CM-5's network makes it possible to partition the machine such that two different partitions do not physically share any part of the network. By preventing users from sending messages to nodes in a different partition (this is implemented in the NI) and by gang-scheduling jobs within each partition (implemented by the operating system), different jobs cannot interfere with each other. To simplify gang-scheduling, the CM-5 network provides a mechanism that allows the operating system to drain the network of all messages from one job before context-switching to another job.

On shared-memory machines, the operating system can use the protection mechanism of the virtual memory system to keep different jobs from accessing each other's memory. However, as long as nodes are allowed to generate an arbitrary number of communication requests, it is still possible for one job to slow down another job's progress by causing congestion in shared portions of the interconnect.

Protection is largely independent of the tradeoffs we study in this dissertation, so we do not further explore the issue.

2.3 Network Interface

The *network interface* ("NI" for short) provides the *interface* that the processing nodes use to *inject* and *extract* messages into and out of the network. More than any other component, the network interface defines the *communication architecture* of a parallel machine. Depending on the NI, the machine depicted in Figure 2.1 may be a message-passing machine, a remote memory access machine (also sometimes called NUMA for Non Uniform Memory Access), or a cache-coherent shared-memory machine. The complexity of the NI varies dramatically depending on what communication architecture it implements.

Message-Passing

For a message-passing communication architecture, the NI can be very simple. For example, the NI on the message-passing Intel iPSC/860 consists of little more than two FIFO buffers and some control circuitry. The *receive* FIFO accumulates incoming data from the network and the *send* FIFO holds data that is to be injected into the network. The processing node accesses the network interface through a set of memory-mapped NI registers. Reading and writing these registers allows the CPU to inject data into the *send* FIFO and extract data from the *receive* FIFO. Other NI registers hold information about the status of the FIFOs or allow the processor to control the NI's mode of operation; e.g., whether or not to interrupt the CPU when the *receive* FIFO fills up. A traditional message-passing library (like Intel's NX [Pierce 88] or Thinking Machines' CMMD [TMC 92]) can be implemented on top of these primitives to provide higher levels of abstraction to the programmer. Some machines include DMA hardware to speed transfer of data between main memory and NI. Generally, DMA operations have to be initiated by the CPU for each packet sent or received, therefore such DMA support is less useful for small packets.

Remote Memory

In a remote memory architecture, processors communicate by accessing a (physically distributed) shared memory space through `load` and `store` instructions. A reference to data that resides on a remote node automatically generates a message to read or write the desired remote data; references to local data are directly satisfied by the node's own memory. To implement such a communication architecture, the NI becomes somewhat more complex. For example, the NI needs to observe the processor's memory references

(e.g., by snooping on the processing node's memory bus), and it must be able to determine, based on the observed address, whether data resides on the local node or a remote node. A simple solution distributes memory among nodes such that the high order bits of a physical address indicate the processing node that holds that memory location.[2] For remote accesses, the NI then needs to create a message, inject it into the network, and possibly stall the processor until the operation is complete. The NI must also be able to access the processing node's main memory in order to reply to memory requests from other nodes.

Cache-Coherent Shared Memory

Cache-coherent shared-memory machines use essentially the same communication interface as remote-memory machines. However, the NI now includes *caches* (or uses the node's main memory as a cache) that can hold data from remote nodes, and the NI must implement some *cache coherence protocol* to keep all caches in the system consistent with each other.[3]

In our work, we examine two classes of cache coherence protocols: *write-invalidate* and *write-update*. In a write-invalidate protocol, a processing node that is about to write to a cache line needs to first *invalidate* all other copies of that cache line, to ensure it has an exclusive (and hence writable) copy. This involves sending invalidation messages to all other nodes that hold a copy of the cache line, and possibly waiting for acknowledgement messages to indicate that the invalidation has been performed. In a write-update protocol, more than one node may keep a writable copy of a cache line. Whenever the cache line is written, the changes are forwarded to all other nodes holding a copy of the line.

The choice of interconnect may influence the design of the cache coherency protocol. For example, if the network supports efficient broadcast operations, then a *snoopy* cache coherence protocol (e.g. [McCreight 84]) can be used. Otherwise, *directory-based* protocols (such as [Censier & Feautrier 78]) are more attractive. . However, the difference

---

[2] The Cray T3D uses an approach similar in spirit. However, the T3D is supposed to scale to large numbers of nodes, and sacrificing enough high-order physical address bits to encode that many processing node numbers would reduce the available per-node physical address space by too much. Instead, the T3D dedicates a smaller number of high-order address bits to encode a node identifier, and each processing node uses a lookup table (the "TLB Annex") [MacDonald & Barrusio 94] to translate this identifier into a full node number.

[3] Such as design usually requires a very tight coupling between the NI and the processing node's memory system, so one could argue whether the caches are part of the NI or part of the processing node. For the purpose of this discussion, we consider the caches for remote data a part of the NI.

between snoopy and directory-based protocols does not affect the programmer's view of the communication interface, so we do not explore this otherwise very important issue further.

In most existing shared-memory machines, all data in the system's caches is backed by main memory. In the case where main memory is physically distributed with the processing nodes, e.g. on the Stanford DASH [Lenoski et al. 92], each cache line has a *home* node, namely the processing node holding the portion of main memory that backs the cache line. To access data in a given cache line, processing nodes send their requests to the home node, which keeps track of all copies of the line and can properly serialize accesses to the cache line.

A different approach is taken in so-called COMA (Cache-Only Memory Architecture) shared-memory machines, such as the DDM [Hagersten 92a] or the KSR-1 [KSR 92]. These systems do not include any "main memory"; data exists *only* in the caches, and cache lines do not have a home node — in other words, there is no *fixed* node in the system that at all times keeps track of a given cache line's state or location, or that serializes accesses to the line. Generally, accessing data therefore involves some form of *search* for a copy of the cache line. In the case of the KSR-1, the search mechanism takes advantage of the inherent broadcast capabilities of the underlying interconnect, which has a ring topology.[4]

In our experiments, we will study both COMA and more conventional cache-coherent shared-memory machines, and we will examine both write-invalidate and write-update protocols.

2.4 Summary

The hardware components of a parallel computer can be organized into three groups: an interconnection network, and a set of processing nodes and network interfaces. We have described some of the design tradeoffs for each of these components.

Our goal is to study the interaction of communication architecture (i.e., the communication interface available to the programmer or compiler) and programming languages. We focus on the design of the network interface, which more than any other component defines a machine's communication architecture.

Issues of programming node and network design are largely orthogonal to our studies.

---

[4] The KSR-1 can actually use two levels of rings, and it maintains directories at the up/down-links between the levels, to determine whether a request needs to be propagated to the other rings as well.

All NI designs we will discuss in this dissertation are intended to work well with off-the-shelf processors. Unless otherwise stated, we assume in our experiments that the underlying interconnection network is *reliable* (e.g., it does not lose, duplicate, or corrupt packets without signalling an error) and provides *FIFO ordering*.

Chapter 3

DATA-PARALLEL LANGUAGES

We have chosen C* [Rose & Steele Jr. 87, TMC 90], a *data-parallel* language, as the high-level parallel language for our experiments. There exists a significant body of work on C* as well as other data-parallel languages such as Force [Jordan 87], Dino [Rosing et al. 90], Kali [Koelbel & Mehrotra 91], Vienna Fortran [Chapman et al. 92] or High-Performance Fortran (HPF) [HPFF 93], to name just a few. This reflects the growing popularity of this type of language. The following two properties are shared by almost all (imperative) data-parallel languages.

- Parallelism is obtained by performing similar (or identical) operations in parallel on the elements of a large data set. The elementwise addition of vectors would be a trivial example.

- The language semantics offer the *illusion* of *lockstep execution*; in other words, no matter how many processors are used, conceptually there is a single "program counter." This execution model avoids race conditions and thus greatly simplifies the understanding and debugging of data-parallel programs.

While the data-parallel model of execution is not as general as arbitrary MIMD computation, it is nonetheless a very powerful model. A study by Fox [Fox 88] showed that 70 out of 84 scientific applications studied, or over 80%, fit the data-parallel model. Klaiber and Frankel [Klaiber & Frankel 93] have demonstrated that even an application that intuitively does not seem to fit the data-parallel model — a distributed event-driven simulation — can be expressed cleanly and efficiently in a data-parallel language.

Several data-parallel languages, including an older version of C* [Rose & Steele Jr. 87], originated as a programming language for SIMD machines. However, researchers have shown [Hatcher & Quinn 91, Rosing et al. 90, Koelbel & Mehrotra 91, Chapman et al. 92] that data-parallel programs can be compiled to run efficiently on MIMD hardware as well — both distributed-memory and shared-memory architectures. To do this, the compiler generates code that includes communication and synchronization operations

carefully chosen to preserve the lockstep semantics of the data-parallel language while also minimizing the total amount of communication.

Compilers for data-parallel languages essentially produce SPMD (single-program, multiple-data) style code, a common programming paradigm on MIMD machines. Parallelizing compilers for sequential languages, such as Paradigm [Su et al. 93], generate similar code. In fact, since most parallelizing compilers exploit mainly data parallelism (e.g., by parallelizing loops), we expect programs generated this way to exhibit execution behavior similar to compiled programs written in high-level data-parallel languages.

In Section 3.1, we give an overview of the C* language, and we draw a brief comparison between C* and High Performance Fortran (HPF) in Section 3.2, demonstrating the similarity of the two languages.

3.1 The C* Language

Several languages featuring data-parallel execution have been proposed over the last years. For this work, we chose the "new" revision of C* [TMC 90, TMC 91a] as a representative data-parallel language. As we will see shortly, C*'s communication operations are typical of what we would expect from other data-parallel languages, hence our findings should extend to those languages as well. We particularly highlight the similarity between C* and HPF in the next section.

Significant work has been done on the compilation of an older version of C* (defined in [Rose & Steele Jr. 87]) for both distributed-memory and shared-memory multiprocessors; see for example [Hatcher & Quinn 91] for a summary. For a detailed description of the current language, the reader is referred to [TMC 91a]; we give a brief overview here.

C* distinguishes between *scalar* and *parallel* variables; the latter have a *shape* associated with them that describes how the data is organized. Attributes of a shape are its rank, layout and number of *positions*; there is one *virtual processor* (VP) per position. Shapes serve as templates to declare parallel variables of that shape: when a parallel variable of some shape S is declared, one element of the variable is allocated in each position (i.e., each VP) of the shape. Parallel variables of identical shape are laid out identically, meaning corresponding positions of all parallel variables of a given shape are mapped to the same VP.

The compiler and run-time system create the final data distribution by mapping the virtual processors onto the physical processors. C* does not provide mechanisms for explicitly specifying this mapping, though the programmer can provide hints. Unfortunately,

there is no way to specify the alignment of two *different* shapes with respect to each other.

To operate on parallel data, the programmer selects a *current shape* using the `with` statement. Simple operations such as addition, when applied to parallel data, are executed in parallel for each position in the current shape.

The following example first declares the shape `MatrixShape` as a two-dimensional (100×100) grid, and then declares two parallel variables, `a` and `b`, of type `double` and shape `MatrixShape`. Since `a` and `b` have the same shape, corresponding positions of the two variables are co-located in the same virtual processor. I.e., the virtual processor at position (i, j) in the shape will hold the matrix elements $a_{i,j}$ and $b_{i,j}$. When data is co-located on the same virtual processor, it is also co-located on the same physical processor, hence the elementwise addition performed in the example below does not generate any inter-processor communication.

```
shape [100][100]MatrixShape;
double:MatrixShape a, b;
with (MatrixShape)
    a = a + b;
```

Control flow in C* is sequential, i.e., from the programmer's point of view, conditional branches, procedure calls, etc. are followed by all processors. In fact, virtual processors behave as if they were executing code synchronously. Parallel operations can be *contextualized* inside a `where` statement by specifying a parallel boolean expression that determines which virtual processors are "active". Conceptually, a `where` statement first executes the `where` branch, then the `else` branch, with parallel operations restricted to the virtual processors on which the condition evaluates to `true` and `false` respectively.[1] The following example computes the square root of the absolute value of each element in the matrix `a`.

```
with (MatrixShape)
    where (a >= 0.0) b = sqrt(a);
        else b = sqrt(-a);
```

Communication in C* is performed by *send* or *get* operations, which are written as parallel *left index* expressions, using a syntax reminiscent of array references. The code below

[1] The semantics of C* specify that scalar code inside branches of a `where` statement is *always* executed, i.e., independent of the condition in the `where`. This is in keeping with C*'s "global model of execution" [TMC 91a].

transposes matrix a by sending each element at position (i, j) to position (j, i). Matrix b is transposed using an equivalent (but not necessarily equally efficient) *get* operation. The parallel expression `pcoord(`i`)` evaluates to a position's index along dimension i of the current shape, i.e., for C*'s row-major layout, `pcoord(0)` yields an element's row number in the matrix and `pcoord(1)` yields the column number. Send and get operations are atomic in that they behave as though all data elements were sent simultaneously.

```
with (MatrixShape) {
    [pcoord(1)][pcoord(0)]a = a;
    b = [pcoord(1)][pcoord(0)]b;
}
```

In the case where multiple VPs send data to the same destination VP, C*'s *combining send* operations allow the programmer to specify a binary combining operation, such as addition, which will be performed on all data arriving at the same destination VP. By default, the receiving VP may arbitrarily choose one of the conflicting data items.

In addition, C* provides powerful *reduction* operations by overloading the C language's "embedded assignment" operators.[2] A simple example that computes the sum of all elements in matrix `a` is given below.

```
double elementSum;
with (MatrixShape)
    elementSum = (+= a);
```

Finally, C* provides access to individual elements of parallel data, which may require data to be broadcast to all nodes. The syntax is the same as a *get* or *send* operation, but all left indices are *scalar* rather than parallel expressions (note that it is trivial for a compiler to detect this). The code fragment below divides all matrix elements by the element in the upper left corner of the matrix. Since the matrix is distributed across nodes but all nodes need the value of $a_{0,0}$, that matrix element needs to be broadcast.

```
with (MatrixShape) {
    a /= [0][0]a;
}
```

[2] More complex reduction and scan operations are available using function call syntax, i.e., the language provides no special operators for them.

Functions in C* can take parallel arguments and return parallel results. For example, the parallel version of the `sin` function is declared as

```
double:current sin(double:current);
```

where `current` is a reserved word referring to the shape that is in effect at the time of the function call. The syntax for calling parallel functions is the same as that for scalar functions; e.g. to compute the sine of all non-negative elements in matrix `a`, one would write

```
with (MatrixShape)
    where (a >= 0.0) a = sin(a);
```

Given the synchronous model of execution, C* programs do not exhibit race conditions, and execution does not depend on nondeterministic events such as message arrival orders.[3] The simple programming model of sequential control flow coupled with deterministic execution makes programming and debugging of C* programs almost as easy as for purely sequential languages. While requiring that all communication be explicit in the source code does place additional burden on the programmer, it also provides the programmer with a clear performance model, exactly *because* any potentially expensive communication operations are clearly visible in the code. One drawback of C* (or similar languages) is that sometimes the synchronous semantics overly constrain a solution, as one cannot efficiently express arbitrary asynchronous operations. Naïve compilation may exacerbate this problem, but some language extensions and compiler techniques can alleviate the problem [Klaiber & Frankel 93].

3.2 C* versus HPF

The C* language originated as a language for SIMD architectures, so there may be concerns over how well it represents other data-parallel languages. In this section, we compare C* with the more recently developed High Performance Fortran (HPF), another data-parallel language. We show that, despite their different syntax, heritage and emphasis, both C* and HPF share important characteristics, especially regarding communication. We provide

[3] The one obvious exception is the *combining* `send` operation. However, it is easy to make this a deterministic operation as well — in fact, Thinking Machines' C* implementation does just that.

this comparison as a "proof-by-example" that our findings obtained from analyzing C* benchmarks and compilation should extend to other data-parallel languages.

As we have seen, variables in C* are explicitly declared parallel by associating a *shape* with them. We can think of the shape as a set of virtual processors, with each virtual processor holding one element of the parallel variable. The parallelism in operations on parallel variables is implicit in C* — the programmer specifies no looping construct or index variables. For example, the code below declares two matrices `a` and `b`, and performs elementwise addition in parallel.

```
shape [100][100]MatrixShape;
float:MatrixShape a, b;
with (MatrixShape) {
    a = a + b;
}
```

In contrast, parallel variables in HPF are first declared as arrays, and then distributed across processors. In HPF, data distribution is a two-step process: the programmer can specify how to distribute the array over a set of *abstract processors*, which the compiler (possibly in cooperation with the run-time system) then maps onto physical processors. HPF's abstract processors are similar to virtual processors in C*. The key difference is that in C*, the mapping of data to VPs is fixed — exactly one element per VP. To operate on parallel variables, HPF provides a parallel looping statement, `FORALL`. Assuming there are 10 abstract processors, the above C* example could be written in HPF as follows:

```
        REAL a(100,100), b(100,100)
!HPF$   PROCESSORS procs(10)
!HPF$   DISTRIBUTE (BLOCK,*) ONTO procs :: a, b
        FORALL (i=1:100, j=1:100) a(i,j) = a(i,j) + b(i,j)
```

Certain forms of parallel operations can be expressed more conveniently using the Fortran 90 triplet notation. For example, we could replace the `FORALL` loop in the above code by the statement

```
a(:,:) = a(:,:) + b(:,:)
```

HPF's approach to data distribution is more complex and flexible than the one taken by C*, but the end result is the same: elements of a parallel variable are distributed among a

set of physical processors. The biggest drawback of C* is probably that one cannot align different shapes with respect to each other. The C* benchmarks we study do not require particularly elaborate data distributions, and would not benefit from HPF's added flexibility. Also, researchers have recently had great success in deriving data distributions *automatically* [Su et al. 93]; this may obviate HPF's complex data distribution specifications.

To express communication, C* and HPF again use different syntax but very similar semantics. C* uses *left index* expressions to describe communication operations, whereas in HPF, any array index operation potentially causes communication. For example, the C* code to set `b` to the transpose of matrix `a` (using a `send` operation)

```
with (MatrixShape) {
    [pcoord(1)][pcoord(0)]b = a;
}
```

has the following equivalent in HPF:

```
FORALL (i=1:100, j=1:100)
    b(j,i) = a(i,j)
END FORALL
```

Like C*, HPF provides reduction operations, and accessing individual elements of a parallel variable (either through the Fortran 90 `SPREAD` operation or through array indexing) may require broadcast operations. While HPF does not include an operation corresponding to the C* *combining* send, other communication operations are semantically comparable, and we argue that the two languages' demands on the communication substrate should likewise be similar.

Since C* parallel operations and HPF `FORALL` loops are semantically equivalent, and since communication operations are also comparable in both languages, the overall compilation strategy and communication requirements for C* and HPF are essentially the same.

We conclude that despite different heritage and emphasis, there are many similarities between HPF and C*, including the computation and communication model. Therefore, we postulate that the findings we present in chapters 5 and 6 will apply to data-parallel languages other than C* as well (at the very least to HPF).

3.3 Summary

The C* language originated as a language for SIMD architectures, whereas more recent data-parallel languages, such as High Performance Fortran (HPF), have been designed from the start with MIMD target machines in mind. We have compared C* with HPF. Though the latter places a much heavier emphasis on allowing the programmer to specify details of how to distribute parallel data, we have found that the two languages are in fact very similar. Specifically, we argue that the demands on the communication architecture are comparable, hence the results (obtained from C* benchmarks) we present in the core of this dissertation should also apply to HPF, and likely to other data-parallel languages as well.

Chapter 4

THE C* COMPILER

In this chapter, we outline our compilation strategy for C*. Our C* compiler is based on a recent version of the compiler by Hatcher and Quinn [Quinn et al. 88, Hatcher et al. 91, Hatcher & Quinn 91] that we have modified for our purposes. The overall approach is to translate C* into mostly machine-independent C code that makes calls to run-time libraries for all communication and synchronization operations. The resulting code is then linked with architecture-specific run-time libraries.[1] Figure 4.1 outlines the compilation process. The simulation framework, used to gather our measurements, is described in section 5.3.

Our C* compiler generates code for an abstract *communication model* which the run-time libraries map onto the target machine's communication architecture. The compiler's communication model need not be identical to the target communication architecture (in fact, it may help to hide some details of the hardware), but of course there must exist an efficient mapping from the former onto the latter. In Section 4.1, we contrast existing communication models for shared-memory and distributed memory targets; our compiler uses a hybrid of these models. In Section 4.2, we then describe in more detail the different tasks that our compiler performs, and we give a very high-level outline of the run-time libraries.

The experiments described later in this dissertation focus on communication operations rather than efficient code generation for *local* (i.e., non-communication) operations. In fact, we explicitly ignore time spent performing computation, so we need not concern ourselves with "traditional" optimizations or C*-specific optimizations (such as reducing the cost of emulating virtual processors) that improve execution speed of local operations.

---

[1] In practice, one would presumably inline most of the communication operations in order to obtain better performance, but this was not necessary for the purpose of this dissertation.

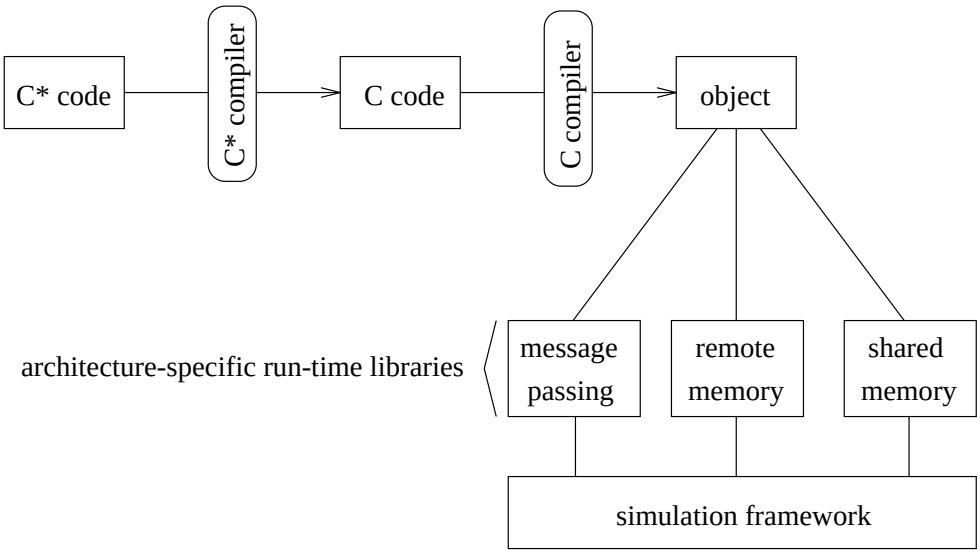


Figure 4.1: Compiling C* for multiple target architectures.

4.1 Alternative Communication Models

Hatcher and Quinn describe two alternative approaches for compiling C* for shared-memory and distributed-memory targets, respectively [Hatcher & Quinn 91]. In this section, we briefly review and contrast their approaches, and then outline the compilation strategy of the C* compiler used in this dissertation.

4.1.1 Compiling for a Shared-Memory Target

In a shared-memory architecture, all nodes can access each other's memory through `load` and `store` instructions. When generating code for such an architecture, the compiler need not treat accesses to remote data any differently from accesses to local data. This greatly simplifies how the compiler handles C* communication operations, since C* `get` and `send` operations translate directly into `load` and `store` instructions. However, the compiler must insert explicit synchronization operations in the generated code to prevent *race conditions*. Hatcher and Quinn describe this approach in [Hatcher et al. 91].

Consider the C* code in Figure 4.2a, where each VP sends a copy of its `y` variable to its left neighbor, which stores the data in its `x` variable. To simplify the following discussion,

we ignore effects at the ends of the VP array, and assume one virtual processor per physical processor, i.e., each physical processor "owns" one element of the parallel variables `x` and `y`.

Figure 4.2b represents the corresponding C code generated for a shared-memory machine. This code is executed independently on all processing nodes; `self` contains the processor number. The parallel variables have been transformed into arrays located in the shared address space,[2] and the C* `send` to variable `x` has been translated directly into a variable access. Note that the compiler had to insert a call to `Synchronize` in order to ensure that processors do not read "their" element in `x` until it has first been written by their neighboring processor. To preserve the inter-node data dependencies in our example, `Synchronize` must make the calling processor wait until its right neighbor has also called `Synchronize`.

As this example shows, the main task when compiling for a shared-memory architecture is to determine a minimal set of points where synchronization is required.

4.1.2 Compiling for a Distributed-Memory Target

For distributed-memory targets, message-passing has traditionally been the communication model of choice (e.g., [Chapman et al. 92, Su et al. 93, Quinn et al. 88]). By analyzing the accesses to parallel variables, the compiler (or the run-time system) determines the set of data elements that must be communicated between nodes, and emits matching pairs of message-passing `msg_send` and `msg_recv` calls to perform the communication.

An advantage of this approach is that the message-passing model combines data transfer and synchronization in one operation. Hence, the processors need not perform synchronization as a separate operation, as is the case in a shared-memory communication model. Also, the generated code can take advantage of the "just-in-time" delivery semantics of messages: a node can send data to another node even before the receiver has posted a matching `msg_recv` operation. The message-passing library (or operating system) on the receiver will simply buffer the incoming message until it is needed by the application program. As a result, there is more "slack" in the synchronization between the two nodes, which may help hide temporary load imbalances, and increases potential for overlap of

[2] Note that unless the cache line size is $\leq$ the size of an integer, a data distribution of one array element per processor is going to cause *false sharing*. We will describe shortly how our compiler addresses this problem.

```
shape [N_proc]S;
int:S x,y;
[.-1]x = y;
use(x);
```

(a) original C* code.

```
SHARED int x[N_proc], y[N_proc];
x[self-1] = y[self];
Synchronize();
use(x[self]);
```

(b) code generated for shared-memory model.

```
int x[1], y[1];
msg_send(self-1, &y[0], sizeof(int));
msg_recv(self+1, &x[0], sizeof(int));
use(x[0]);
```

(c) code generated for message-passing model.

```
int x[1], y[1];
remote_write(remote_addr(self-1, "x[0]"), y[0]);
Synchronize();
use(x[0]);
```

(d) code generated for remote-memory model.

Figure 4.2: Communication for different communication models.

communication and computation.

Returning to the C* example in figure 4.2, the compiler determines that each physical processor sends one data item to its left neighbor, and receives one data item from the right neighbor. Again, we are assuming a mapping of one VP per physical processor, and we ignore effects at the ends of the VP array. Figure 4.2c shows the C code a compiler would generate for a message-passing communication model. Since the target is a distributed-memory machine where each node has its own separate address space, the compiler physically partitions the parallel variables: each processor holds one element of the variables `x` and `y`. The `msg_send` and `msg_recv` operations implement the communication by sending a copy of `y` to the left neighbor (`self-1`) and receiving a new value for `x` from the right neighbor. Note that the code does not require an explicit synchronization operation. Instead, a processor can safely continue execution as soon as the data from the right neighbor has arrived.

Unfortunately, the message-passing communication model also has several drawbacks. First, traditional message-passing incurs run-time *protocol* overhead, e.g., for managing the buffers required for the "just-in-time" delivery semantics. Felten has shown that protocol overhead degrades the performance of message-passing codes [Felten 93a]. Since the C* communication operations read and write parallel variables that are already allocated by the compiler, most of the buffer management overhead is completely unnecessary.

Second, in a message-passing model of communication, data transfer and synchronization are *always* combined, even when only one of the two functions is needed. Since neither function is free, the processors have to perform unnecessary work; we discuss this point in more detail in Chapter 6.

Third, programs often exhibit communication patterns that cannot be analyzed at compile time. In that case, the compiler cannot determine the set of data elements that must be communicated, and hence it cannot generate the required `msg_send` and `msg_recv` operations. Note that this problem does not occur in the shared-memory model. The reason is that communication in the shared-memory model is *requester* based, meaning a given node can read and write another node's memory *without* the cooperation of the other node. The compiler therefore can simply emit code to perform the read or write operation. In the message-passing model, *both* nodes must actively participate in the communication: the sender must execute a `msg_send`, and the receiver must execute a matching `msg_recv`. To generate code for a given node A, the compiler must therefore have complete knowledge

about *all* requests for A made by *any* of the other nodes. A general solution to this problem requires extra communication and computation at run-time, just to determine which nodes have to send which data items where [Clark et al. 92, von Hanxleden et al. 92]. To reduce this overhead, the run-time system can analyze a communication pattern once at run-time, and amortize the cost of the analysis over many reuses of the pattern [Wu et al. 91]. Also, researchers have optimized the analysis phase itself [Leung & Zahorjan 93].

However, we wish to reduce this overhead further, even in cases where the above techniques are not applicable.

4.1.3 Our Compilation Strategy

We now describe the strategy our compiler uses to generate code for shared-memory and distributed-memory target architectures.

Distributed Memory

As we have seen above, compilation for a message-passing model is harder than for a shared-memory model, and programs that cannot be fully analyzed at compile-time may incur additional run-time communication overhead. To make things worse, traditional message-passing libraries already have a very high per-message overhead.

One reason for this is that messages are a very general mechanism for communication. Conceptually, a message *per se* has no semantics attached, and it is up the the receiver to place an interpretation on the message, and to determine how to process it. The semantics of message-passing also specify many functions besides the actual data transfer — e.g., automatic buffering of messages and implicit synchronization between sender and receiver.

However, most communication in the C* language only requires one specific primitive operation to be performed, and there is only a small number of commonly used primitives. For example, almost all C* communication operations involve reading or writing memory on a remote node.

Our C* compiler therefore uses a *remote memory access* model of communication which better matches the language's communication operations. The remote memory model is very similar to the shared-memory model, in that the two fundamental communication primitives are reading and writing of memory on a remote node. To access data on a remote node, the compiler need only generate an operation on the requesting node. Since remote-memory access communication does not perform implicit synchronization between sender

and receiver, the compiler has to insert explicit synchronization operations to preserve inter-node data dependencies — just like it does for the shared-memory model.

In fact, as far as our compiler is concerned, compilation for the remote-memory and shared-memory models is exactly the same, since the machine-specific communication libraries hide the few remaining differences.

Figure 4.2d shows the code generated for the remote-memory access model; note the similarity between this version and the one for the shared-memory model. The expression `remote_addr(nodeno, "var")` computes the address of variable `var` on node number `nodeno`, for use in the `remote_write` operation. In order to obtain good performance, nodes must be able to compute this address quickly. Fortunately, this is not a problem for most regular data distributions; for irregular distributions, *all* compilation approaches will incur higher overhead for address and index calculation.

Note that the remote read and write operations can be implemented efficiently on a message-passing communication architecture using a low-level mechanism like Active Messages [von Eicken et al. 92], which provide a very low overhead data transfer mechanism, without the protocol overhead of message passing.

On the other hand, we have introduced explicit synchronization operations that were performed implicitly in the message-passing model. We can think of the synchronization operations as another form of protocol overhead, taking the place of the "traditional" message-passing protocol overhead [Felten 93a, Felten 94]. At this point, we make no attempt to quantify this tradeoff. In Chapter 6, we will see that the true advantage of compiling for a remote-memory model is that it uses a small set of simple, well-defined communication primitives. This will allow us to provide simple hardware support for communication.

In some cases, the compiler can *recombine* data transfer and synchronization, and eliminate some of the explicit synchronization operations. Our compiler does not currently perform this optimization, so the results presented in the remainder of this dissertation have slightly higher synchronization traffic than what we could expect from a production quality compiler.

Note also that there is ample opportunity for combining data transfer and synchronization in the run-time libraries, e.g. inside `broadcast`, `reduce` and `barrier` operations. Even though the compiler generates code for a remote-memory model, the run-time libraries for the message-passing communication architecture do combine data transfer and

synchronization wherever possible.

Shared Memory

We make one small change to Hatcher and Quinn's compilation strategy for shared-memory, namely we use a different layout for parallel data. Allocating parallel variables contiguously in memory may increase false sharing and distort our results. For example, even operations that only access locally owned data (such as elementwise addition) generate inter-node traffic due to false sharing if the per-node partition of a parallel variable is not a multiple of the cache line size.

Our compiler allocates disjoint memory areas for each processor and maps each processor's partition of parallel data into the processor's own memory area. In effect, we simulate a distributed-memory machine on the shared-memory hardware — with one crucial difference: all data is still accessible as part of the shared memory space.[3]

A production-quality compiler would use other techniques to reduce false sharing, such as padding variables or otherwise changing the layout [Su et al. 93, Ju & Dietz 91, Eggers & Jeremiassen 91]. Yet even with our rather ad-hoc method, we were able to significantly reduce the level of false sharing in our benchmarks. Also, both the shared-memory and remote-memory models use the same kind of data layout, which makes comparisons between the two easier. The drawback of this approach, like most more sophisticated data layout strategies, is that array index calculation becomes more complicated.

4.2 Compiler Overview

We now describe in more detail the C* compiler used in this dissertation. The compiler must perform three major tasks:

- generate code for parallel computation and manage the virtual processor model
- generate code for inter-node communication
- insert synchronization to maintain inter-node data dependencies.

[3] The original C* compiler by Hatcher and Quinn stored parallel variables as contiguous arrays in shared memory [Hatcher & Quinn 91]; it therefore did not address the issue of false sharing.

4.2.1 Parallel Computation

We first describe how to generate code for parallel statements (except communication), and how the virtual processor model is managed.

Parallel variables are implemented as arrays, with one entry for each position in the parallel variable's shape; the arrays are then distributed among the physical processors. The C* *shape* of a variable is represented at run-time by a *shape descriptor*, which contains information such as the total number of positions in the shape, how the data is distributed among the physical processors, and how many array elements are mapped to a given processor. In our compiler, the number of physical processors need not be known at compile-time, since the data distribution for all shapes is computed at run-time. Consider the declarations

```
shape [1024]S;
int:S a;
```

When the program containing these declarations is executed on 16 processors, each physical processor will map 64 positions of the shape `S`. The descriptor for shape `S` is initialized to reflect this data distribution.[4] Each processor then allocates an array of 64 integers to hold its part of the parallel variable `a`. The compiler generates roughly the following C code to be executed on each node.

```
ShapeDescriptor S;  /* this node's copy of shape descriptor */
int *a;             /* this node's portion of variable 'a'  */
/* initialize shape descriptor: each node has 64 positions  */
S.posnsTotal = 1024;
S.posnsThisNode = 1024 / N_nodes;  /* 1024/16 = 64 per node */
/* allocate this node's portion of parallel variable `a'    */
a = stack_alloc(S.posnsThisNode * sizeof(int));
```

Note that we assume that each processor gets its own copy of the variables `S` and `a`; on some shared-memory machines such variables must be marked "private".

We compile parallel statements by wrapping a *virtual processor emulation loop* (VP loop for short) around them. For example, the parallel statement

---

[4] Each physical processor maintains its own copy of all shape descriptors.

```
with (S) {
    a = 0;
}
```

is compiled in to the following C code:

```
currShape = &S;
for (vp=0; vp < currShape->posnsThisNode; ++vp) {
    a[vp] = 0;
}
```

Note the assignment to `currShape`, which keeps track of C*'s *current shape*. If the number of physical processors (and hence the details of the data distribution) were known at compile-time, the expression `currShape->posnsThisNode` could be replaced with a constant, e.g., 64 for execution on 16 processors.

Sequences of parallel statements of identical shape can of course share a single enclosing VP loop. However, intervening scalar operations, such as an assignment to a scalar variable, must only be executed once and hence cannot be nested inside the VP loop. The compiler must "break" the VP loop at the location of the scalar operation.

The cost of breaking VP loops can be greater than immediately obvious. Recall that in C*, every parallel statement is implicitly "contextualized" by any (dynamically) enclosing `where` statements. The VP loop must therefore include a test to check whether the VP being simulated in a given iteration is active. When the compiler breaks a VP loop into multiple parts, it may also have to introduce a temporary variable to hold the context. For example, the sequence of statements

```
where (a == 0) {
    a = 1;
    scalar = 99;
    a = 2;
}
```

generates the following (unoptimized) code:

```
/* allocate temporary to save context across VP loops */
temp = stack_alloc(currShape->posnsThisNode * sizeof(BOOLEAN));
/* first part of VP loop  */
```

```
for (vp=0; vp < currShape->posnsThisNode; ++vp) {
    temp[vp] = (a[vp] == 0);        /* save the context     */
    if (temp[vp]) {                 /* this VP active?      */
        a[vp] = 1;
    }
}
/* scalar statement; splits VP loop in to parts */
scalar = 99;
/* second part of VP loop  */
for (vp=0; vp < currShape->posnsThisNode; ++vp) {
    if (temp[vp]) {                 /* reuse the context    */
        a[vp] = 2;
    }
}
stack_unwind(temp);
```

Note the introduction of the `temp` array which is required to save the current context across VP loops.

Various optimizations can improve execution speed. For example, the compiler could move the scalar operation before or after the VP loop, promote the scalar to a parallel variable, or evaluate the conditional expression in each VP loop fragment, assuming this is cheap enough and data dependencies permit it (in our simple example, there exists an inhibiting data dependency on variable `a`). In [Klaiber & Frankel 93] the authors discuss other optimizations that reduce the cost of VP emulation.

We reiterate that the experiments described later in this dissertation focus on communication operations rather than efficient code generation for *local* (i.e., non-communication) operations. In fact, since we explicitly ignore time spent performing computation, the optimizations discussed above do not affect the results we present in chapters 5 and 6.

4.2.2 Inter-Node Communication

By analyzing the usage of the C* communication operations in the source code, the compiler can classify them as one of five *communication primitives*:

- `get` — Each VP retrieves one data item from another VP.

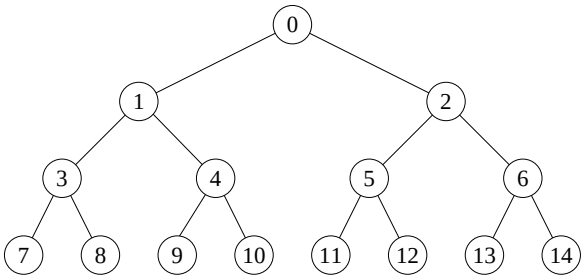


Figure 4.3: Fanin/fanout tree of processing nodes.

- `send` — Each VP sends one data item to another VP.
- *combining* `send` — As above, but with combining for collisions.
- `broadcast` — Data from one VP is broadcast to all other VPs.
- `reduce` — Each VP contributes one data item to a global sum.

In addition, the compiler inserts `barrier` synchronization operations to preserve inter-node data dependencies, this is the sixth communication primitive.

Each of the six primitives is implemented in a machine-specific library function. For example, the shared-memory implementation of `get` uses processor `load` instructions to access data; the remote-memory implementation uses active messages. From the compiler's point of view, the `barrier` primitive is the only one that performs any synchronization between nodes.[5]

Our run-time libraries implement the `broadcast`, `reduce` and `barrier` primitives using *fanin/fanout* tree algorithms. We arrange the machine's processing nodes in a k-ary tree, as shown in Figure 4.3 for $k = 2$. In a `broadcast`, the node owning the data to be broadcast sends the data to the root (node 0) of the tree. Starting at the root, each node forwards the data to its children. In a `reduce` operation, each node sends its contribution to its parent node. The parent computes the local reduction of its data and the data obtained

[5] When a `reduce` is followed by a `barrier`, the compiler merges the two, since both already use the same communication pattern.

from the children, and sends the result to its parent. Once the reduction reaches the root of the tree, the final result is `broadcast` to all nodes. Broadcasts, reductions and barriers can therefore be performed in $O(logp)$ operations on p processors. Note that the tree structure need not be physically implemented in the interconnection network; we simply impose it logically.

Each step in the fanin/fanout tree requires data transfer as well as synchronization — a node cannot, for example, forward broadcast data to its children until it has received the data from its parent node, as well as notification that the data has arrived. Our run-time libraries combine the two functions, data transfer and synchronization, into a single communication operation wherever the underlying communication architecture allows this.

The quality of the code generated by our compiler could easily be improved, for example, we could inline most of the communication in the shared-memory architecture, eliminate some data copying inherent in the run-time libraries, or reduce the cost of index calculation, e.g., by applying strength reduction. However, as mentioned at the end of Section 4.2.1, our experiments do not measure the cost of local computation, so this does not affect our results.

4.2.3 Synchronization

As outlined above, our compiler inserts explicit synchronization operations to preserve inter-node data dependencies, even on distributed-memory machines. (Note that when compiling for a shared-memory machine, this step is always necessary.) The overall approach is rather simple. The compiler uses data-flow analysis to determine where inter-node data dependencies may arise, and inserts synchronization operations to cut the dependencies.

Figure 4.4(a) shows a C* code fragment that has inter-node data dependencies between statements 1 and 3, and 2 and 4: the parallel variables `x` and `y` are written by one processor before they are read by another processor. Figure 4.4(b) shows that a single synchronization barrier is enough to properly synchronize the generated code, since it cuts both dependence arcs $1 \rightarrow 3$ and $2 \rightarrow 4$.

The minimal set of barriers can be found in linear time for sequential code, but for general control flow graphs, this is an NP-hard problem [Hatcher et al. 91]. Our compiler uses a set of heuristics to achieve near-optimal barrier placement in practice, e.g., it places barriers before rather than inside loops whenever possible.

As a side-effect of inserting explicit synchronization operations to preserve inter-node

```
0.  int:S x,y;
1.  [.-1]x = ...;
2.  [.+1]y = ...;
3.  use(x);
4.  use(y);
```

(a) original C* code.

```
int x[1], y[1];
remote-write(remote-addr(self-1, "x[0]"), ...);
remote-write(remote-addr(self+1, "y[0]"), ...);
BARRIER;
use(x[0]);
use(y[0]);
```

(b) properly synchronized remote-memory model code

Figure 4.4: Preserving inter-node data dependecies.

data dependencies, the run-time system can use asynchronous messages to implement the remote read and write operations — the communication operations do not need to complete until the next barrier synchronization point. In effect, this is a software implementation of a relaxed memory consistency model, similar to release consistency [Gharachorloo et al. 90].

Several optimizations can further reduce the cost of synchronization. First, for some communication patterns that are amenable to static analysis, point-to-point synchronization can be used instead of barrier synchronization. For example, when data is exchanged between neighbors in a linear array, global barrier synchronization is too "strong;" synchronization between pairs of neighboring nodes is sufficient. If the required number of point-to-point synchronizations is small enough, this approach achieves lower synchronization latencies than barriers. However, when each node needs to synchronize with more than two other nodes, the number of messages sent for synchronization increases, since barrier synchronization only sends two messages per node.

Second, the compiler could recombine separate data transfer and synchronization operations, which may improve performance if the underlying communication system can support it.

While our compiler does not perform the above optimizations, the run-time communication libraries offer many opportunities for combining data transfer and synchronization, and our run-time libraries for the message-passing architecture do in fact take advantage of this optimization.

4.3 Summary

We have outlined the compilation of C* for shared-memory and distributed-memory architectures. When compiling for a distributed-memory target, we express communication in terms of remote memory accesses instead of message exchanges. This approach, similar to Hatcher and Quinn's compilation strategy for shared-memory machines [Hatcher et al. 91], avoids most of the overheads associated with traditional message-passing.

The more significant advantage of our approach is that communication is performed by a small set of well-defined primitives which are much less general than message-passing. In Chapter 6, we will see how this allows us us to provide simple and efficient hardware support for C* communication.

Chapter 5

ARCHITECTURAL COMPARISON

In this chapter, we study communication architectures for data-parallel programs by comparing three existing models: message-passing, remote-memory and cache-coherent shared-memory. Our goal is to gain a better understanding of how well these architectures support data-parallel programs, and to evaluate the underlying communication costs inherent in each of the three architectures.

A fair evaluation of such fundamentally different architectures has so far been difficult to produce for several reasons. First, the communication model greatly influences the programming model (and vice-versa). Programs are typically hand tailored for different communication architectures, often resulting in vastly different algorithms for the same application. Second, while program execution time can be measured on different multiprocessors, such measurements are difficult to compare, since the many *implementation* differences between machines — such as processor architecture and cycle time, memory system details, and bus technology — tend to obscure the *architecture-inherent* differences that we are interested in.

We avoid this problem by a two-pronged approach: first, we begin with a *single* suite of benchmarks written in C*, which are compiled to the architectures under consideration. We thus measure the work required by each architecture to execute the *same* data-parallel programs. Second, we examine metrics that are mainly *technology-independent*. For example, we evaluate the data and control traffic that flows over the interconnect for various benchmarks, the traffic due to synchronization, and the number of network round-trip latencies incurred.

Our methodology is as follows. We compile each of our C* benchmarks using a C* compiler. Our compiler generates code that is more or less machine independent, and includes calls to a runtime library that manages all inter-processor communication. We have written an optimized runtime library for each architecture. For example, the library simply executes `load` and `store` instructions for a shared-memory machine (simulating the cache and bus as necessary), while simulating message sends and receives for a message

passing machine. More details of our methodology are presented in Section 5.3.

This approach has three advantages. First, the benchmarks represent a compiler-generated workload, which as outlined earlier, we believe is indicative of future workloads. Second, and perhaps more importantly, this allows us to make meaningful comparisons across a range of machines, as all execute the same source program. Third, our measurements are concerned with more abstract metrics such as traffic and network round-trips, not cycle time, and are thus independent of many implementation details.

Our results quantify the relative interconnect work (e.g., bandwidth consumed, number of messages injected, number of round-trip latencies incurred) required to execute these data parallel programs on different architectures. Those measurements do not by themselves translate directly into performance figures; however they are a key factor, and also point out some relative strengths and weaknesses of the architectures. Given technology-specific parameters, such as message startup cost, bandwidth available, etc., one can derive a first approximation of actual communication cost from our measurements. The results in this chapter should therefore be considered more as a guideline for determining architectural tradeoffs rather than a direct indicator of which of the models is "best." For example, the results show that a significant fraction of the total traffic used by the shared-memory architectures for these benchmarks is explicit synchronization. We argue that those machines would benefit greatly from architectural support, such as adaptive or user-selectable cache coherence protocols [Carter et al. 91, Bennett et al. 92], full/empty bits on memory words [Agarwal et al. 91, Alverson et al. 90], a network or network interface dedicated to barrier synchronization and reduction (e.g., the *control network* on the CM-5 [TMC 91b]), or direct access to message-passing primitives as implemented on the Alewife machine [Agarwal et al. 91].

As far as our latency measurements are concerned, we find that even the basic message-passing machine can easily hide network latencies by using asynchronous message exchanges. To achieve similar results, a shared memory machine needs architectural enhancements such as lockup-free caches, hardware support for prefetching, selective write-update (instead of write-invalidate), relaxed memory consistency, or asynchronous propagation of writes. Without these enhancements (which may require substantial changes in hardware and software), the shared-memory architecture can incur up to an order of magnitude more network latency than the message-passing model.

This chapter is organized as follows. In Section 5.1, we define the baseline architectures

Abbreviation	Description
MSG_32	Baseline message-passing model with maximum packet of 32 data bytes.
MSG_inf	Message passing with a maximum packet of 64 Kbytes.
MSG_blk	Message passing that attempts to aggregate multiple requests into one message, provided they exhibit constant stride.
NUMA	The remote-memory model, with a wordsize of 8 bytes.
CACHE_n	Write-invalidate cache-coherent model, cache line size of n bytes, broadcast-capable interconnect.
CACHE_n_wu	Cache-coherent model with selective use of write-update.
CACHE_n_dash_xxx	Uses a point-to-point interconnect and a DASH-like coherency protocol.

Figure 5.1: Summary of architectural models.

we use in our studies, and in Section 5.2, we outline how the C* communication primitives are implemented on these different architectures. Section 5.3 describes our simulation methodology and Section 5.4 discusses the benchmarks we use. In sections 5.5 and 5.6, we present our measurements of interconnect traffic and latency, respectively, and point out the strengths and weaknesses of the different communication architectures. We discuss related work in Section 5.7 and give a summary of this chapter in Section 5.8.

5.1 Architectural Models

We consider three baseline architectures: message-passing, remote-memory and cache-coherent shared memory. All architectures use 64-bit addresses and 16-bit processor IDs. Table 5.1 summarizes the models used in our study, which are described in more detail below.

Message-Passing Model

In the message-passing model (**MSG** in the rest of this dissertation), all interprocessor communication occurs through explicit message passing. All messages contain a standard

header that includes the destination processor ID and a message type. No fixed format is defined for the body of a message.

We make no assumptions about how messages are injected by the sending processor (operating system calls, user-level calls, or special processor instructions) or received (interrupt, polling by the CPU, or handling entirely in hardware). However, some messages do require the active participation of the receiving processor in order to exploit implicit synchronization information present in the arrival of a message.

We implemented three variations of the message-passing model: MSG_32, which has a maximum packet size of 32 bytes, and MSG_inf, which allows messages of arbitrary size. Another model, MSG_blk, is based on MSG_inf but attempts (at runtime) to aggregate multiple per-VP requests into a single message.

Remote-Memory Model

Our remote-memory (**NUMA**) model is based on architectures such as the BBN Butterfly [Crowther et al. 85] or the Cray T3D [Cray 93]. Memory is statically distributed among processors and there are no caches. Processors access memory through 1, 2, 4, or 8-byte `load` and `store` instructions. When a processor issues a memory reference, the hardware decides whether that reference is to local or remote memory. Local accesses are completed in processor-local memory. For a remote access, the processor creates a request message and injects it into the network. Request messages contain enough information for the remote node to create a reply message.

Cache-Coherent Model

For the cache-coherent shared-memory model (**CACHE**), we examine several different variants. For most of our results in this chapter, we show measurements for a COMA (cache-only memory architecture [Hagersten 92b]) roughly based on the KSR-1 [KSR 92], which assumes an interconnect capable of broadcast.[1] We also simulate an interconnect similar to the DASH [Lenoski et al. 92], where, based on its address, a memory block is assigned a "home" processor that handles all requests for that memory block. The coherency protocol used by the DASH does not require a broadcast-capable interconnect

[1] The KSR-1 uses a hierarchical interconnect consisting of up to two levels of rings. Our model is simplified in that we assume a single-level ring interconnect.

and only sends point-to-point messages, an important consideration for scalability.

We consider both write-invalidate (CACHE) and write-update (CACHE_wu) coherence protocols. In the write-invalidate protocol, a node wishing to write a cache line first acquires an *exclusive* copy of the cache line by *invalidating* all other copies. In the write-update protocol several nodes may hold writable copies; whenever a node writes (part of) a cache line, the changed words are forwarded to all other nodes holding a copy. We assume that a *write-cache* [Jouppi 93] is used, i.e., successive writes to the same cache line generate a *single* forwarding message. For both the write-invalidate and write-update versions, we assume that data is always fetched in cache line units.

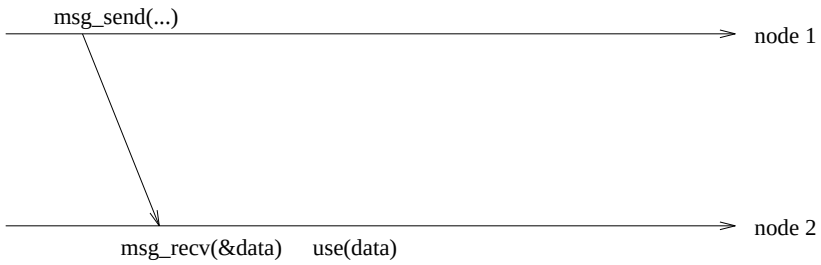
5.2 Implementation of C* Communication Primitives

As noted before, the C* communication primitives are implemented in machine-specific runtime libraries. In this section, we describe how the C* communication primitives are mapped onto the alternative architectures and point out any mismatches between the language's needs and the different communication architectures. For a more detailed description of the C* communication primitives, the reader is referred back to sections 3.1 and 4.1.

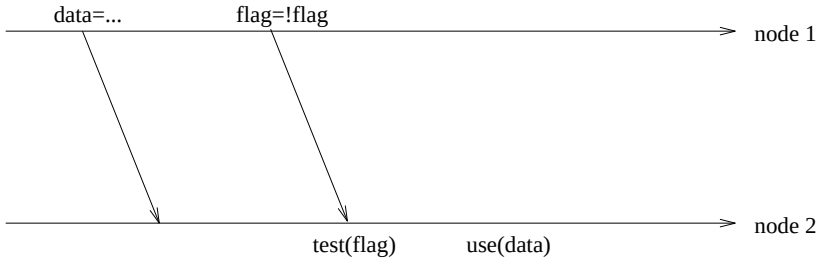
`get` and `send` In the message-passing model, one message is created for each VP in order to send or request data. The NUMA and cache-coherent models simply use `load` and `store` instructions to access remote data and the hardware initiates any necessary communication. If the per-VP data is large, multiple messages are sent, or multiple `load`/`store` instructions are issued.

combining `send` Combining sends are easy to implement in a message-passing model, with the combining operation performed by the receiving node. The other models need to perform the combining operation on the sending nodes, which requires the use of locks to ensure proper serialization among multiple senders.

`reduce` and `barrier` Reductions and barriers are implemented using a fanin/fanout tree. The message-passing model implements the tree operations directly in terms of sending messages, whereas the NUMA and CACHE implementations are based on the lock-free barriers described in [Mellor-Crummey & Scott 91], which achieve the theoretical minimum of $(2p - 2)$ remote accesses on p processors.



(a) Combined data transfer and synchronization in MSG.



(b) Separate data transfer and synchronization in NUMA.

Figure 5.2: Implicit versus explicit synchronization.

`broadcast` Broadcasts are implemented using a fanout tree (essentially the second half of a reduction): the node owning the data sends it to the root of the fanout tree, which then distributes it to all nodes. Note that even though the interconnect in the CACHE model has a broadcast capability, this facility is not available to the user, and hence cannot be used to implement the C* `broadcast` operation.

Implementing fanin/fanout trees on NUMA and cache-coherent machines is more costly than on message-passing machines. Each step in the combining tree requires synchronization, e.g. a parent node needs to know when data from its children is available and vice-versa. In a message-passing environment, this synchronization information is conveyed implicitly with the data transfer, whereas the other models require a separate, explicit synchronization step.

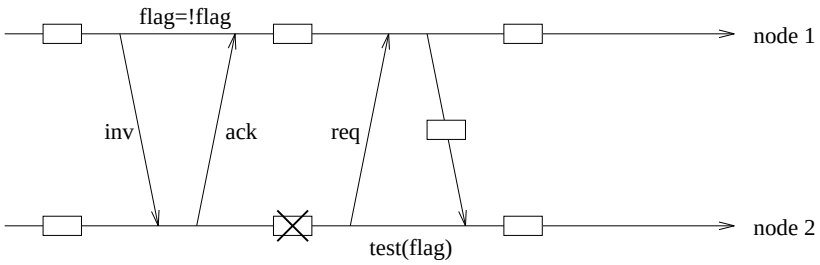


Figure 5.3: Traffic for synchronization under write-invalidate protocol.

Figure 5.2 demonstrates implicit vs. explicit synchronization for two nodes, where node 1 produces data which is to be consumed by node 2. Obviously, this operation requires synchronization in addition to the data transfer, since node 2 must be able to determine when the data has arrived. In the message-passing model (Figure 5.2a), data transfer and synchronization are achieved by sending a single message: the receiver posts a `msg_recv` operation that matches the sender's `msg_send` operation, and the completion of the `msg_recv` call serves to indicate arrival of the data.

In the NUMA model, there is no operation corresponding to explicit receipt of a message, and data transfer cannot be combined with synchronization. Instead, synchronization needs to be performed as a separate operation. In Figure 5.2b, data is "sent" by simply assigning it to a variable, `data`, in node 2's memory. To implement the synchronization, node 1 toggles the `flag` variable, and node 2 busy-waits for `flag` to change before it reads `data`.[2]

The CACHE model uses the same approach as the NUMA model, however the write-invalidate protocol interacts badly with the synchronization operations. Figure 5.3 shows the sequence of operations that take place in the CACHE model for synchronization. The boxes represent copies of the cache line containing the `flag` variable. Before node 1 can write `flag`, it first needs to invalidate node 2's copy of the cache line. This operation results in two messages, one invalidation request and one acknowledgement message.[3] When later, node 2 needs to read the `flag`, another two messages are sent, one request for the cache

[2] Note that this approach relies on FIFO delivery of messages through the network.

[3] Under a weaker memory consistency model, the acknowledgement message could possibly be omitted or executed asynchronously.

line and one reply carrying a copy of the cache line. The same operation that took a single message in the NUMA model now requires four messages, one of which carries a copy of an entire cache line. This results in significantly higher synchronization traffic. Note that by using a write-update protocol for synchronization operations, the number of messages can be reduced to one, as in the NUMA model.

In most benchmarks, the `get` and `send` operations access little data per VP (e.g., a single word), which prevents the message-passing runtime library from taking advantage of large packets. For the MSG_blk model, we implemented a run-time library that attempts to combine *multiple* small per-VP requests into one large request, provided the accesses exhibit constant stride. This mimics compiler optimizations that aggregate messages for fixed communication patterns. We will see in Figure 5.5 that message aggregation can significantly reduce traffic in the message-passing model, for some benchmarks.

Recall that, as discussed in Section 4.1, we assume that only `barrier` operations carry synchronization information. For example, when the compiler generates a `send` call, it does not require the library to ensure that data dependencies are preserved, or to perform any kind of synchronization between the sender and receiver. This allows an arbitrary number of `get`, `send`, or `broadcast` operations to be outstanding at any time, and lets us amortize synchronization operations over multiple data transfers.

5.3 Simulation Methodology

As discussed in Chapter 4, the C* source is compiled into mostly machine-independent C code, with communication primitives provided by machine-specific run-time libraries. To simulate the message-passing architectures, we instrument the run-time libraries to capture the communication and synchronization operations, since these are the only points where inter-processor communication occurs.

In contrast, the cache-coherent and NUMA models require that all memory references be traced. To this end, our compiler instruments the generated code and tracks all references to parallel variables. References to scalars are always local since they are replicated on every node, therefore we ignore them. In our simulations of the cache-coherent models, we assume fully-associative, infinite-size caches, thus eliminating conflict and capacity misses. We also detect all cold misses and exclude them from the results as well. Thus, all inter-node traffic is due to active sharing of data. In our benchmarks, the data layout was chosen (by programmer and/or compiler) such as to reduce false sharing as well. Note that these

are very optimistic assumptions but they also isolate the results from more implementation details, such as cache size or organization. For the results presented in this chapter, keep in mind that in reality, the CACHE model would generate significantly more traffic.

Statistics are gathered through execution of the benchmarks on a KSR-1; we drive several simulation models at once to amortize the cost of program execution. Traffic is divided into three categories:

- Traffic due to `get` and `send` operations, in essence point-to-point communication between pairs of nodes.

- Traffic due to `broadcast` and `reduce` operations, which are examples of collective communication operations.

- Traffic entirely due to synchronization, such as barriers or the explicit synchronization in CACHE and NUMA fanin/fanout tree operations.

The first two categories are further subdivided into *data* sent and *control* information, such as message headers or invalidation messages. All measurements in this dissertation were performed on simulations of 8-node machines. We have simulated selected benchmarks for up to 64 nodes and verified that our conclusions remain essentially the same.

5.4 Benchmarks

For our experiments, we utilized six benchmarks: `jacobi`, `gauss`, `matrix`, `ocean`, `shallow`, and `misra`. `Jacobi` implements 18 passes of a Jacobi iteration on a 128×128 grid, using a two-dimensional block partitioning of the grid. Remote data is accessed with `get` operations, and a reduction is performed at the end of every iteration to determine an error term.

`Gauss` performs Gaussian elimination with pivoting on a 128×128 matrix which is distributed by rows. `Matrix` multiplies two 128×128 matrices which are distributed by columns. Both `matrix` and `gauss` broadcast matrix columns or rows, respectively. `Gauss` also uses reduction operations to determine pivot rows.

`Ocean` is a model of ocean circulation using a one-dimensional data partitioning. Data is accessed exclusively through `get` operations accessing 8- and 16-byte quantities. The

original Fortran code came from the Ocean Engineering department at Oregon State University.

`Shallow` is based on Fortran from NCAR. It is an atmosphere model based upon the shallow-water equations. A 64×64 grid is distributed by columns and data is accessed using both fine-grain (one `double` at a time) `get` and coarse-grain (one column at a time) `send` operations.

Finally, `misra` is an event-driven logic simulator. Essentially all traffic in this application is due to reduction and combining `send` operations. The combining `send` communication patterns are highly irregular and fine-grained.

5.5 Traffic Measurements

In this section, we discuss the results of our traffic measurements. To simplify the presentation, Section 5.5.1 starts by selecting, for detailed examination in the rest of the chapter, a line size for the cache-coherent model and one version of the message-passing architecture.

In the rest of this section, we present our measurements of *interconnect traffic*. As mentioned earlier, we divide traffic into three major categories: traffic from `get` and `send` operations, traffic from `reduce` and `broadcast` operations, and pure synchronization traffic. In sections 5.5.2 through 5.5.4, we examine each traffic component separately and then, in Section 5.5.5, compare the relative importance of the individual traffic components. We also contrast traffic in the KSR-like and DASH-like models and examine how increasing the number of processors affects our results, in sections 5.5.7 and 5.5.8, respectively.

In this chapter, we focus exclusively on the interconnect traffic generated by the different communication architectures. While traffic is an important metric, it obviously does not translate directly into performance. The amount of work to send a message also varies drastically between models — in the CACHE and NUMA models, a single `load` or `store` instruction suffices to have the hardware generate a message, whereas in existing message-passing machines, the cost is significantly higher. We will address the issue of message handling overheads in more detail in the next chapter.

5.5.1 Selecting Simulation Parameters

While we have abstracted away many implementation details such as timing, cache size and organization, bus technology, etc., our baseline models still require some parameterization.

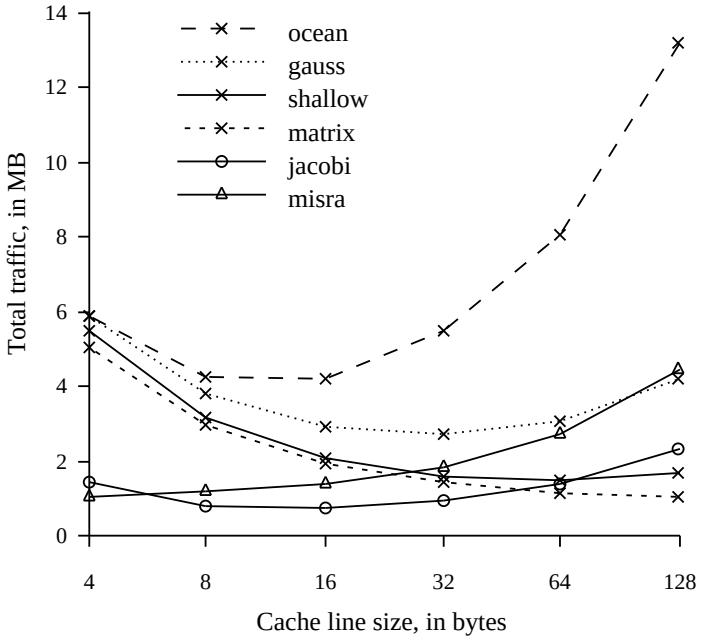


Figure 5.4: Total traffic in CACHE model, as function of cache line size.

Presenting all possible variations would be impractical; in this section we select, for detailed examination in the rest of the chapter, a line size for the cache-coherent model and one version of the message-passing architecture.

Choosing a Cache Line Size

Obviously, the performance of a cache-coherent architecture depends to a high degree on the cache line size chosen. Longer cache lines take better advantage of spatial locality, but may result in higher levels of false sharing or inefficient use of the interconnect when only parts of a cache line are actually used.

Figure 5.4 shows total traffic for the invalidation-based cache coherent model as a function of cache line size. From these results, we choose a cache line size of 32 bytes as the basis of our comparisons throughout the rest of this chapter.[4]

Observe that `ocean` is not very well optimized for spatial locality. We include this

[4] Further measurements indicate that this is a good line size for the other variants (write-update, or DASH-style protocol) of the cache-coherent model as well.

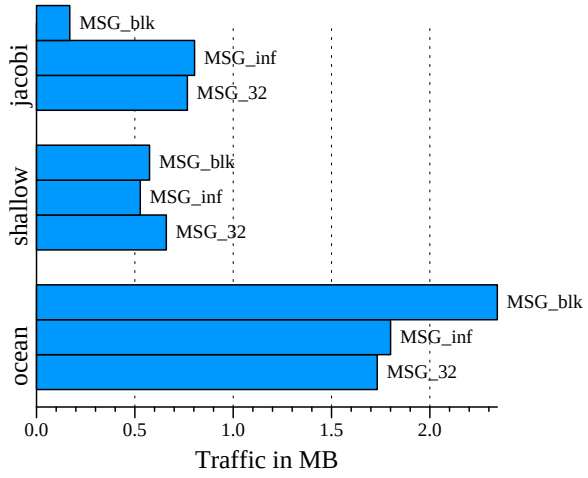


Figure 5.5: Message-passing models: total traffic.

benchmark as an indication of what happens when a compiler fails to arrange memory accesses for optimum cache utilization.

Choosing a Message-Passing Implementation

Figure 5.5 shows the total message traffic for our three message-passing models. The baseline model, MSG_32, allows a maximum of 32 data bytes per packet and thus uses the same granularity as the CACHE model. Larger messages must be split into individual packets, each carrying its own copy of control information. Two variants of the baseline model attempt to reduce this overhead. The MSG_inf model allows packets up to 64 KB, and can therefore communicate large per-VP data structures in a single message. Since `get` and `send` operations often generate many small requests (one per VP), the MSG_blk model attempts to aggregate at runtime multiple per-VP requests of constant stride into one large packet. Both of these optimizations come at a cost: the MSG_inf model needs a larger "size" field in the message header, and the MSG_blk model needs to send count and stride information.

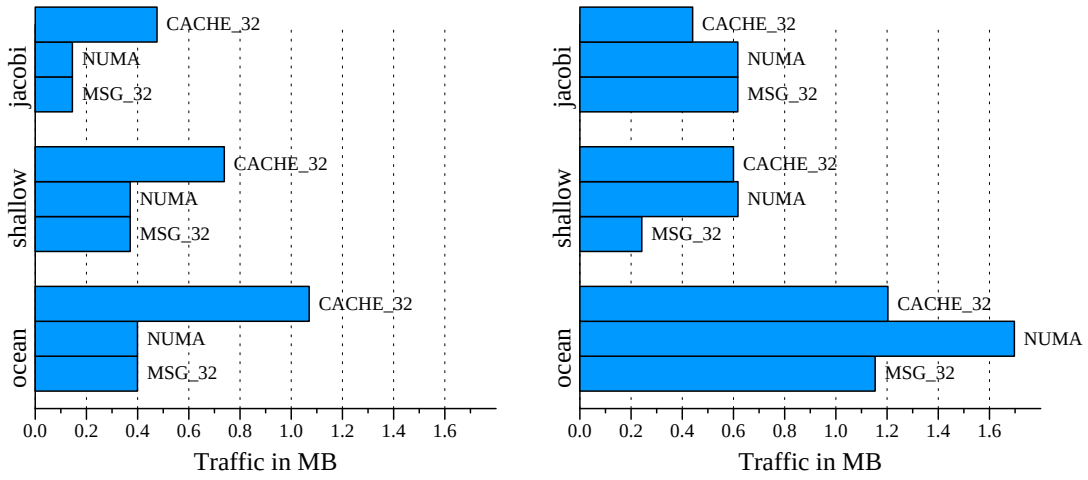
We see in Figure 5.5 how these changes affect total traffic. In the case of `jacobi`, each physical processor holds a subgrid of VPs, and each of the VPs on the subgrid's boundary issues a request for a small data item. Therefore, aggregating messages (MSG_blk) reduces

traffic by amortizing the message header over more useful data. However a single per-VP request fits perfectly into a small packet and hence simply increasing the maximum packet size (MSG_inf) only adds control information — the packet header in MSG_inf contains 2-byte *size* field whereas in MSG_32, 1 byte is sufficient. In the case of `shallow`, which uses a coarser data decomposition, only one VP per physical processor sends data off-node. Each item represents an entire column of data, therefore it helps to send larger packets. Since only one VP per physical processor sends data, aggregation does not improve performance beyond that, however. Finally, in `ocean` only one VP sends data per physical processor, and data items are small. Therefore, neither larger packets nor aggregation can reduce control traffic.

Throughout the rest of this chapter, we use the simpler MSG_32 model as the basis of our comparisons, although for some regular communication patterns or coarse-grained data decompositions, its performance may be improved considerably.

5.5.2 *Traffic from* `get` *and* `send` *Operations*

We now turn to evaluating the traffic generated by the various communication architectures.



(a) Data traffic (b) Control traffic

Figure 5.6: Comparison of `get`/`send` traffic.

Figures 5.6a and 5.6b show the *data* and *control* traffic, respectively, generated by the C* `get` and `send` operations for a set of representative benchmarks. Control traffic includes

packet headers and "pure" control messages, such as requests for data or invalidation messages. As our CACHE model sends much less control information per message (i.e., CACHE uses much smaller message headers) than the other models, we also show the *number* of messages due to `get` and `send` operations in Figure 5.7.

All data transfers in the CACHE model occur in cache line units. In the case of `jacobi`, which uses a two-dimensional block decomposition, half of the remote accesses have unit-stride and touch all the data in a cache line, while half use only one data item per cache line fetched. This results in a more than threefold increase in data traffic over the MSG and NUMA models. `ocean` also suffers from this problem.

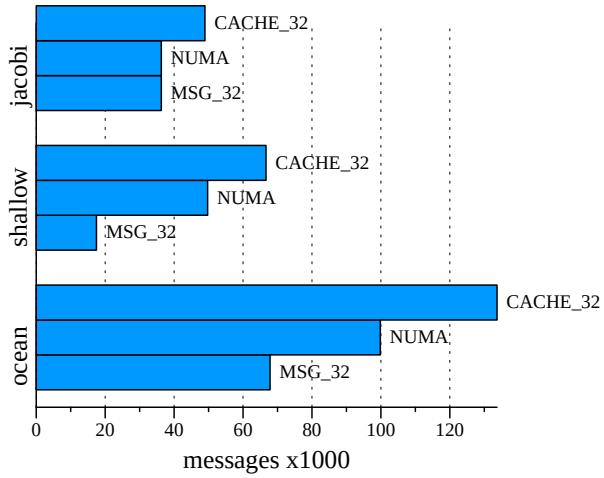


Figure 5.7: Number of messages for `get`/`send` operations.

A different effect is responsible for higher data traffic in `shallow`. Some variables in this application are alternately written locally and from a remote node, via a `send` operation. The two write operations together require four messages in the CACHE model, two of which carry control information only, and two of which carry a copy of the cache line.[5] The MSG and NUMA models only require one message for the remote write, and none for the local write. This explains why despite good cache line utilization, the CACHE model generates nearly twice the data traffic of the MSG and NUMA models for `shallow`.

Figure 5.6b shows that the CACHE model generates 30% less control traffic than the

[5] If it is known in advance that the entire cache line is going to be overwritten, then the first transfer of data can be omitted. However, this information is not always easy to infer.

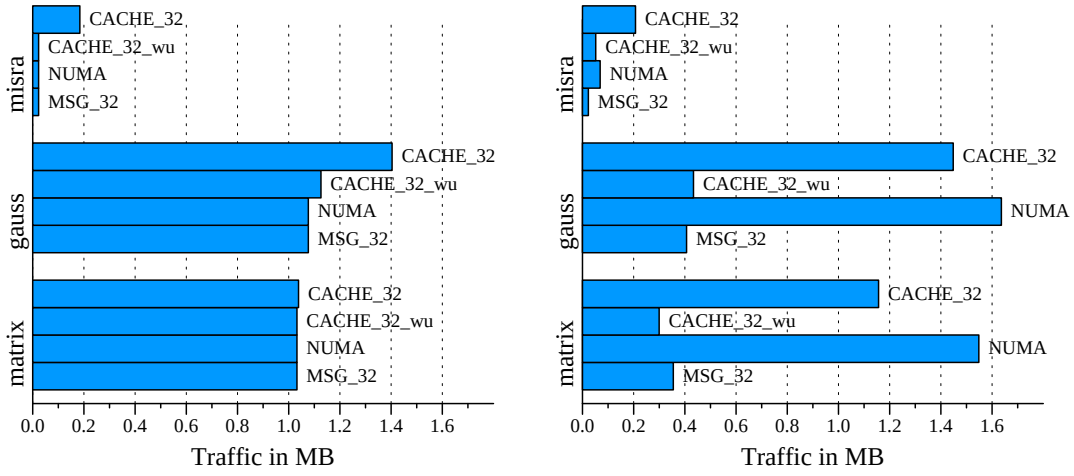
others for `jacobi`, although it actually sends a larger number of messages — about 30% more than the MSG model, as shown in Figure 5.7. As a node attempts to write grid values locally, it must first invalidate the copies of boundary elements that have been accessed by remote nodes, causing extra control messages. However, because the CACHE model sends very little control information per `load` or `invalidate` request, the actual traffic is lower. The behavior of `ocean` is similar, and in the case of `shallow`, extra control messages are needed to implement the `send` operation, as discussed above.

In summary, the CACHE model relies heavily on spatial and temporal locality to amortize the cost of the cache coherence protocol. The former is needed to make full use of all data in a cache line, and can often be achieved through careful optimization of data layout. However, many scientific applications, especially iterative algorithms, tend to update most if not all of their data set on each iteration; therefore, little temporal locality exists between iterations, which renders caching less useful. For example, in `jacobi`, a processor accesses each neighboring grid point exactly once per iteration; the values for those grid points will be overwritten by the remote processor before the next iteration. In other words, data from remote nodes are not reused and hence caching of remote data does not pay off — however, the system still incurs the overhead of the cache coherence protocol. As a result, the CACHE model generates 2 to 3 times the data traffic of the MSG model for the applications shown in Figure 5.6.

The NUMA model often suffers from the fact that it can only access one word (64 bits) per request and therefore may require large amounts of control traffic — about 45% to 150% more than MSG on the `ocean` and `shallow` applications, respectively.

5.5.3 Traffic from `broadcast` *and* `reduce` *Operations*

To illustrate performance on `broadcast` and `reduce` operations, we choose the `matrix`, `gauss` and `misra` benchmarks. `Matrix` broadcasts columns of a matrix (128 words), whereas `misra` performs many reduction operations on small (4-byte) data items. In `gauss`, both broadcasts of matrix rows and reductions of small data items are used, so its behavior lies between the other two benchmarks. Figures 5.8a and 5.8b show data and control traffic, respectively, for these benchmarks. Since the basic CACHE model performs rather badly on the communication patterns in the fanin/fanout tree, we also consider the CACHE_wu model, which allows the library to request a *write update* protocol for selected `store` instructions.



(a) Data traffic (b) Control traffic

Figure 5.8: Comparison of `bcast`/`reduce` traffic.

As `matrix` transfers large contiguous chunks of data, the CACHE model can fully utilize the cache line, and all three models transfer about the same amount of data.

At the other extreme, `misra` does not use broadcasts and all data traffic shown here is due to reductions of small data. Since the CACHE model transfers data in cache line units, this results in an eightfold increase in data traffic over the MSG and NUMA models, which only transfer the actual amount of data needed. Note that the write-update cache-coherent model, CACHE_wu, does not have this problem; it performs as well as the MSG and NUMA models.

Control traffic is also strongly influenced by granularity and sharing patterns. In `matrix`, entire columns of the matrix (128 elements) are broadcast at once. The CACHE and MSG models transfer 32 bytes at a time, whereas the NUMA model is restricted to one word per request and thus needs to send more requests than the other models. While the CACHE model makes good use of the entire cache line, the write-invalidate protocol also interacts badly with the fanout-tree communication pattern, which explains why CACHE sends over three times as much control traffic as the MSG model. However, due to the large number of small requests it requires, the NUMA model produces even more control traffic than the CACHE model. The same drawback of the write-invalidate protocol is responsible for the CACHE model's tenfold increase in control traffic over MSG on the `misra` benchmark. Note that overall, the CACHE_wu model performs significantly better than CACHE, with

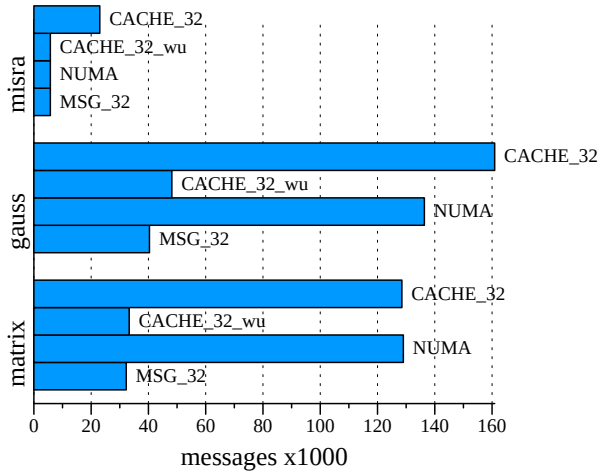


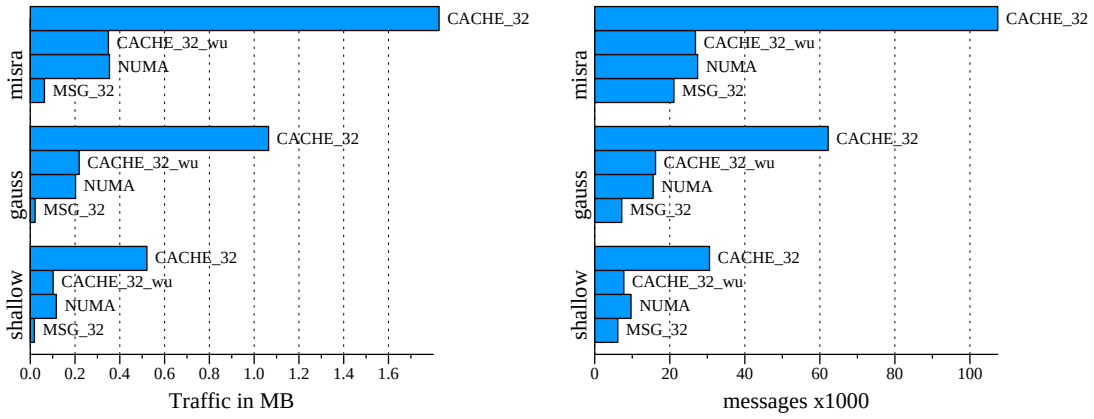
Figure 5.9: Number of messages for `bcast`/`reduce` operations.

performance comparable to the MSG model on the `matrix` and `gauss` benchmarks.

Some of the differences between models are caused by different message header sizes, so we also show the total *number of messages* sent for `broadcast` and `reduce` operations in Figure 5.9. While the ratios between the different models are slightly different than for number of *bytes* sent, our conclusions remain the same.

In summary, we see that implementations of combining tree operations are significantly more expensive in the CACHE model than in either the NUMA or MSG models. Whenever fanin/fanout tree operations are performed, the CACHE models generate excess traffic as cache lines ping-pong between nodes. Considering how important combining tree operations are for scalability purposes, this can be considered a severe drawback of the CACHE model. The CACHE_wu model allows the communication library to request a write-update protocol for selected `store` operations, which improves the performance of the cache-coherent model to the level of the message-passing architecture — at the cost of extra hardware, including the write-cache as described on page 44.

The NUMA model again shows that the lack of a block transfer mechanism drastically increases the amount of control information sent, to over four times that in the MSG model on the `matrix` and `gauss` benchmarks. Both of these benchmarks broadcast data structures significantly larger than a word, namely entire rows or columns of a matrix.



(a) Amount of data sent (b) Number of messages sent

Figure 5.10: Comparison of synchronization traffic.

5.5.4 Synchronization Traffic

Figure 5.10 shows the amount of data and number of messages that are used solely for internode synchronization. Synchronization operations occur primarily in the fanin/fanout trees used to implement barriers, reductions and broadcasts.

A major advantage of the MSG model is that it can combine data transfer and synchronization in a single message, as discussed in Section 5.2. In our benchmarks, the opportunity to combine data transfer and synchronization arises frequently, specifically in all fanin/fanout tree operations. Since the NUMA and CACHE models need to implement the synchronization as a separate operation, it is therefore not surprising that the MSG model generates far less synchronization traffic than the other models.

The CACHE model suffers from additional drawbacks. First, the write-invalidate cache coherence mechanism requires four message exchanges for each synchronization operation. Second, all data is transferred in units of entire cache lines, whereas theoretically, the exchange of a single bit is sufficient to implement the synchronization. As a result, the CACHE model generates between 25 and 30 times the amount of synchronization traffic of the MSG model.

In addition to the CACHE model, we again consider the CACHE_wu model, which allows the library to select a *write update* protocol for `stores` that are use to implement synchronization operations. The CACHE_wu model reduces the cost of each step to a single

message, and transfers written data at a granularity finer than a cache line. Performance of CACHE_wu is about equal to that of NUMA, with a small difference due to handling of locks in `send` operations, which we will not discuss here.

The NUMA model offers better control over data movement than the CACHE model, and allows data transfers in small units more appropriate for the synchronization operations. While both NUMA and CACHE_wu perform noticeably better than the basic CACHE model, they still generate 5 to 10 times as much synchronization traffic as the MSG model.

Figure 5.10b shows the *number of messages* sent for synchronization, which abstracts from the size of message headers and the fact that the CACHE models have to send entire cache lines for synchronization when a single bit would suffice. As we see, the graphs look essentially the same as Figure 5.10a, hence our above conclusions remain the same.

In summary, we re-emphasize that, as noted in the previous section, write-invalidate protocols exhibit significant degradation when faced with a sharing pattern that exhibits no temporal locality. A write-update model fares much better as long as data can be forwarded at a fine enough granularity, i.e., less than a cache line.[6] Combining synchronization with data transfer, as done in the MSG model, is a useful technique for further reducing synchronization traffic.

5.5.5 Contribution of Traffic Categories

To visualize the importance of each traffic category, Figure 5.11 shows the individual traffic components for each model. Clearly, synchronization traffic (shown as the rightmost striped bar in the figure) can represent a major fraction of total traffic for the CACHE and NUMA models. In the case of `ocean` and `misra`, the CACHE model even generates *more* traffic for synchronization than for transfer of actual data. The NUMA model performs synchronization operations more efficiently, since no cache coherence mechanism can get in the way, and data can be transferred in units smaller than a cache line.

In the NUMA model, control traffic is a very prominent component, due mainly to the lack of a block-transfer mechanism: many individual messages, each carrying a full message header, are required to transfer large data structures (e.g. a row of a matrix).

Not surprisingly, the amount of synchronization traffic is negligible in the message-

[6] Note that a write-update model, indiscriminately applied to *all* remote accesses, can result in huge increases in traffic, depending on the access pattern. This is borne out by simulations we performed. In general, therefore, the user should have control over the cache coherence protocol.

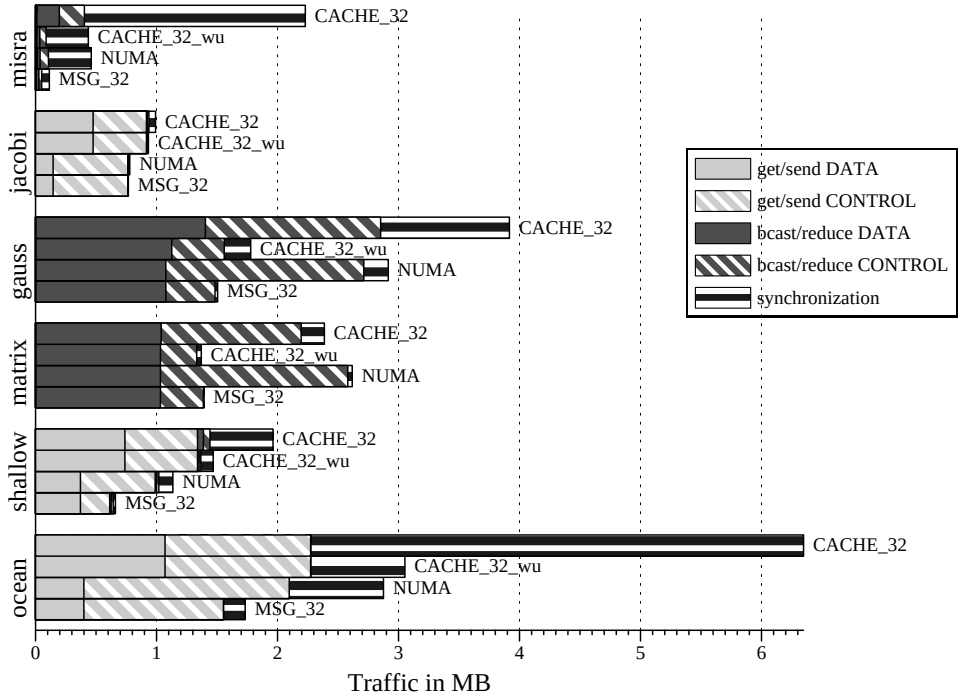


Figure 5.11: Overall traffic.

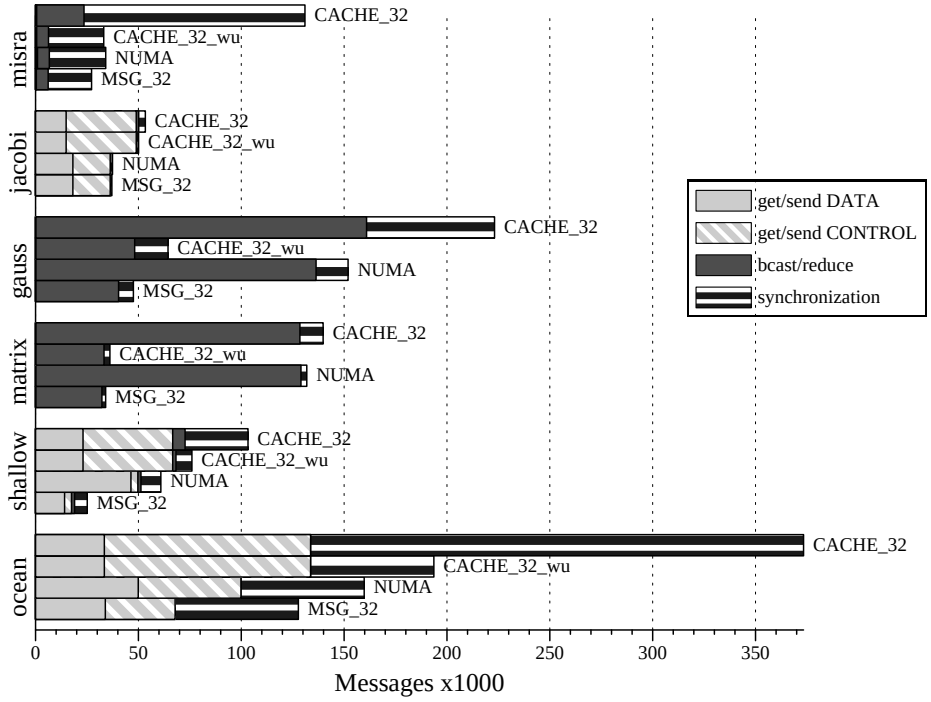


Figure 5.12: Overall traffic: number of messages.

passing model. This shows the advantage of exploiting the synchronization information implicit in the arrival of a message.

Note that no spatial or temporal locality is present in the synchronization operations (only one bit of synchronization information is transmitted, and nothing is gained by caching that bit), therefore large cache lines will drastically increase the amount of synchronization traffic in the cache-coherent models.

We argue that since synchronization traffic can comprise such a large fraction of total traffic in the NUMA and CACHE models, alternative synchronization mechanisms need to be explored for these models. In addition, the NUMA model would benefit greatly from a block transfer mechanism.

5.5.6 Total Number of Messages Sent

So far, we have concentrated on the amount of traffic sent over the interconnect. However, processors also incur some fixed overhead for each message sent or received. Figure 5.12

shows the number of messages sent in each traffic category. The CACHE model sends between 30% to 500% more messages than the MSG model, though the CACHE_wu model performs much better, generating between 6% to 70% more messages than the MSG model (with the exception of `shallow`, where CACHE_wu still sends about three times as many messages as MSG.)

While this difference is significant, recall that the amount of work the CPU performs to handle each message also varies drastically between models — in the CACHE and NUMA models, a single `load` or `store` instruction suffices to have the hardware generate a message, whereas in existing message-passing machines, the cost is significantly higher. We will examine these per-message overheads in more detail in the next chapter.

Figure 5.12 also yields another insight — in the results shown in Figure 5.11, the CACHE model is penalized for being unable to send data at a smaller granularity than an entire cache line. By just counting the number of messages sent, we can abstract away the effects of cache line size. However, even under such optimistic assumptions, the CACHE model still generates more traffic than either the NUMA or the MSG model.

5.5.7 Broadcast versus Point-to-Point Interconnect

In this section, we briefly compare the CACHE model based on the KSR and the one based on the DASH. Recall that the KSR's coherence protocol relies on the interconnect's broadcast capabilities, and that cache lines have no fixed "home" node. In contrast, the DASH's coherence protocol only sends point-to-point messages and each cache line has a home node which handles all requests for that cache line. The latter approach may require more messages to be sent. For example, the KSR always handles a read miss in two messages — one to broadcast the read request and another to broadcast the reply containing the cache line. On the DASH, a read miss may generate three messages if the home node does not have a valid copy of the requested cache line. In that scenario, the requester sends a message to the cache line's home node, the home node forwards the request to some node that currently holds a copy of the cache line, and that node sends a reply message back to the requester.

Similarly, the KSR can invalidate multiple copies of a cache line with a single broadcast message, whereas the DASH must invalidate each copy with a separate message.

Figure 5.13 shows that the DASH-like model generates only about 10%–20% more traffic than the KSR-like model. There are three reasons for this. First, in most of the cases

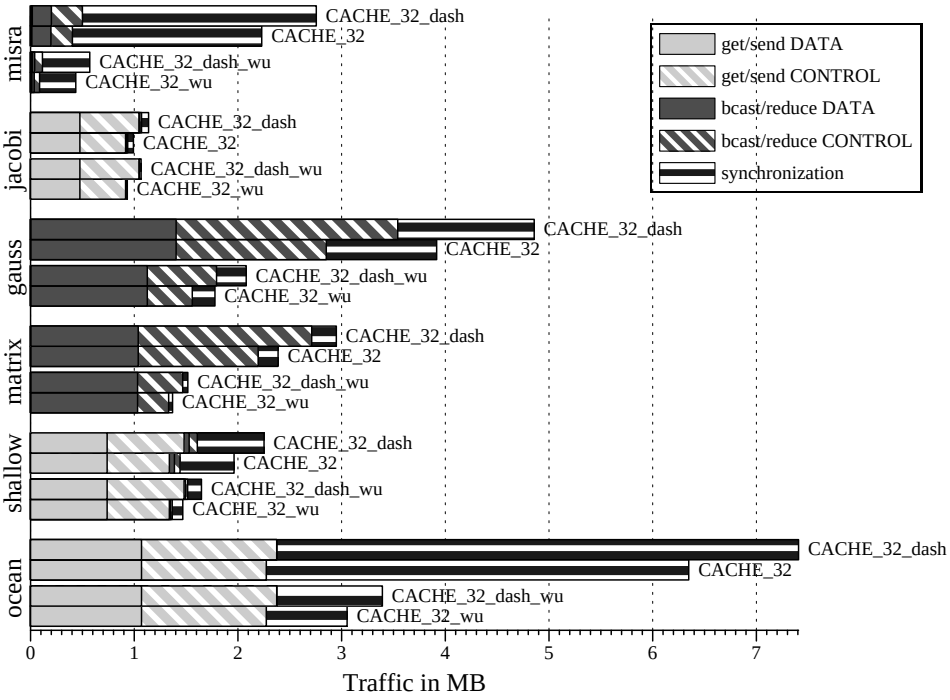


Figure 5.13: Traffic in DASH-like and KSR-like models.

where the DASH sends more messages than the KSR, the extra messages are small control messages (read requests, invalidations, acknowledgements, etc.) that do not carry a copy of a cache line. Second, in our benchmarks the number of nodes sharing a given cache line is usually low, so the DASH does not require substantially more invalidation messages than the KSR. Third, the data layouts used in our benchmarks usually allow the DASH to handle read or write misses in two messages, just like the KSR.

In addition, keep in mind that messages in DASH are point-to-point, whereas they must be *broadcast* in the KSR model. For a given size machine, and assuming equal technology parameters for the interconnect, one would expect the DASH-like machine to perform better, as the aggregate bandwidth required, taking into account number of nodes visited by each message, should actually be lower.

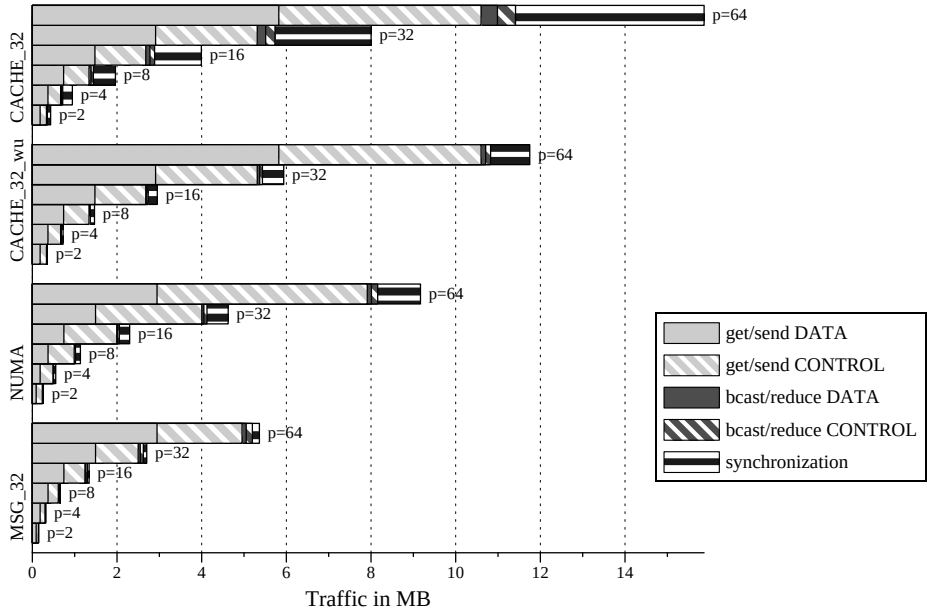
5.5.8 Scaling of Benchmarks

Figure 5.14 shows how traffic for the `shallow` and `gauss` benchmarks increases with the number of processors. As we can see, the relative importance of the different traffic categories remains essentially the same. The scaling behavior of these two benchmarks is representative of our other benchmarks and we therefore expect our conclusions to remain valid for larger numbers of processors as well.

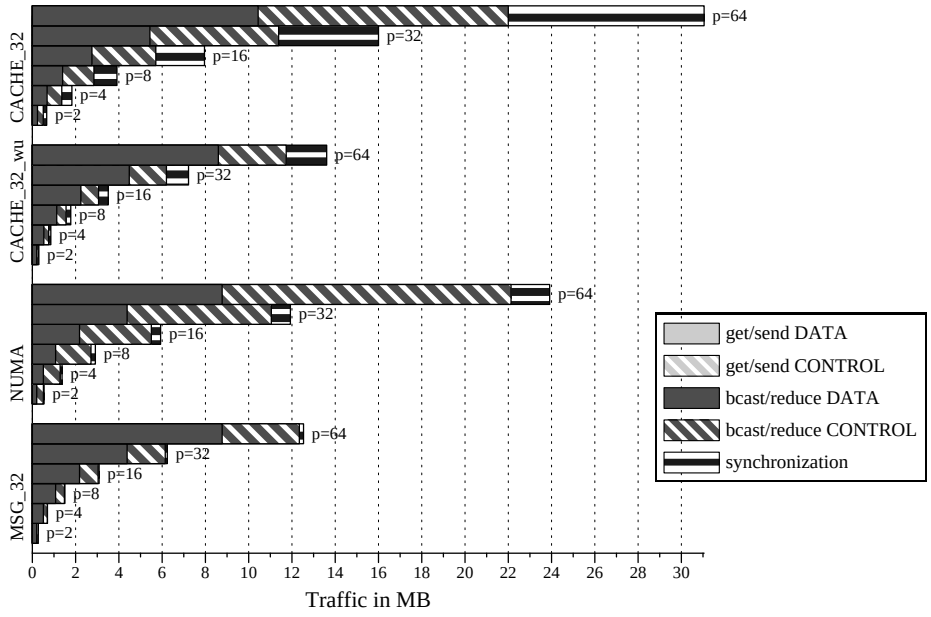
5.6 Latency Measurements

One important aspect of communication performance that we have ignored so far is *latency*, the amount of time that elapses between the initiation and completion of a communication operation. Communication latency can severely limit the performance of parallel machines, as processors may have to busy-wait or *stall* until a communication operation completes. For example, when accessing data on a remote node, the requesting processor may have to wait one round-trip through the network until the desired data is available. This is similar to the latency problem that uniprocessors face as the ratio of processor speed to memory speed increases, except that network latencies are generally an order of magnitude higher than memory latencies.

Most parallel architectures try to reduce the impact of communication latency by providing some form of *asynchronous* (or *non-blocking*) communication primitives. The idea is that a processor can initiate the communication and perform some other work while the



(a) Traffic in `shallow`.



(b) Traffic in `gauss`.

Figure 5.14: Traffic as function of number of processors.

communication is in progress. Depending on how much "other work" is available, this approach can partially or completely hide the latency of the communication operation.

It is easy to implement this approach on a message-passing architecture, where the basic communication primitives — injecting a message into the network, or receiving a message — are inherently asynchronous. For example, the processor can proceed immediately after sending a message.[7] Hence, to retrieve data from another node, a processor can inject a request for that data into the network and proceed with other work before attempting to use that data. Note also the message-passing communication model does not have an architecture-inherent limit on the number of outstanding asynchronous messages. This makes it easier to *pipeline* large numbers of communication operations, which can help to hide communication latency.

The situation is somewhat different on cache-coherent shared-memory machines, where the basic communication primitives, i.e., `load` and `store` instructions, are not as suited to asynchronous communication as the message-passing model's primitives. Even when `load` and `store` are not outright *blocking* (where the processor always stalls until the instruction completes), the number of outstanding requests is usually limited, e.g., to the number of registers available in the processor. Also, allowing fully asynchronous writes may affect the type of memory consistency model that the system provides. Cache-coherent architectures therefore provide latency hiding either through separate mechanisms such as data *prefetching* or by modifying the cache coherence mechanism. For example, the cache coherence protocol could be made adaptive [Archibald 88, Stenstrom et al. 93, Bennett et al. 90, Carter et al. 91], or it could implement a weaker memory consistency model [Hutto & Ahamad 90, Gharachorloo et al. 90].

In this section, we evaluate our three communication architectures with respect to the amount of communication latency they incur. We begin by describing, in Section 5.6.1, the tradeoffs we had to make in our simulations to obtain results that are largely independent of implementation details, and comparable across different architectures. In Section 5.6.2, we examine the effects of augmenting the basic CACHE model with various latency hiding mechanisms and we select the best variation of the CACHE model for further study in the rest of this chapter. As we did before, we divide traffic into three major categories:

---

[7] If the network is busy, some buffering would have to be performed by the sender to keep the processor from stalling. The key point is that *in principle*, barring congestion, the fundamental communication primitives in the message-passing architecture are naturally non-blocking.

traffic from `get` and `send` operations, traffic from `reduce` and `broadcast` operations, and pure synchronization traffic. In sections 5.6.3 through 5.6.5, we examine each traffic component separately and then, in Section 5.6.6, compare the relative importance of the individual traffic components. We also contrast communication latency in the KSR-like and DASH-like cache-coherent models, in Section 5.6.7.

5.6.1 Assumptions and Limitations

As in our traffic studies, our goal is to examine the *technology-independent* differences between the architectures, meaning we are interested in metrics that are unaffected by details such as network topology, routing strategy, or the ratio of processor to interconnect speed.

For example, communication latency is largely determined by the time it takes for a message to traverse the interconnection network, which is a function of the network's topology, its routing strategy, and the technology used to build the routers. Another factor that contributes to latency is the amount of time is takes to *process* a message — for example, the time taken to receive a request for data, access the data, form a reply message and inject the reply into the network.

To allow reasonable comparison of our results across architectures, we again focus on implementation-independent metrics. Specifically, we gauge communication latency by simply counting the *number of message round-trips* during which processors are stalled. This is a reasonable simplification, since we can expect the gap between processing speed and network latency to grow larger in the future, hence the latter is bound to become the dominant factor. Furthermore, we do address the issue of message processing overheads in the next chapter.

Also, as mentioned above, realistic architectures use various techniques for hiding communication latencies, using some form of asynchronous communication, such as prefetching. The extent to which these techniques are successful depends critically on how much computation can be performed between the initiation of the communication operation, and the time that the result of the communication is needed. Specifically, the work used to hide the communication must take time greater than or equal to the communication latency. But, to simulate this effect correctly, we would need to know (at least) the ratio of processor to interconnect speed — which is exactly the kind of implementation detail that we wish to avoid. We therefore make some tradeoffs that sacrifice simulation detail in favor of

implementation independence:

- Realistically, the latency for a given message can vary depending on the distance the message travels, or the amount of network contention it encounters along the way. Both distance and contention depend on, for example, the network topology, speed, and routing strategy — implementation details that we wish to ignore. For our study, we assume that network latency is a *constant*, so we can express communication latencies in terms of the number of message round-trips taken. Essentially, our studies assume a contention-free network with a fully interconnected topology.

- Our programs make frequent use of barrier synchronization. It is clear that the processor stall time in barriers is very sensitive to load balancing — a single processor reaching the barrier "late" can hold up all others. For our studies, we assume that the benchmarks are perfectly load balanced, meaning all processors reach the barrier at the same time. Note that for most of the benchmarks we study, this is actually a good approximation. If there was a high degree of load imbalance, our approach would overestimate the importance of communication latency relative to load imbalance.

- As outlined above, to fully simulate latency hiding, we would need to know the ratio of processor to interconnect speed. This is because the effectiveness of latency hiding depends on the amount of computation that can be done while the communication operation is in progress. However, we can make a *conservative* approximation. When a variable is written by one processor and read by another processor later, the C* compiler inserts a synchronization point between the two accesses, in order to prevent races. When simulating asynchronous writes, we assume that the issuing processor has to wait for one message round-trip at the next synchronization point, i.e., until the last write has been acknowledged. This is a worst-case assumption, but it is reasonable for our benchmarks, where the inter-processor data-dependencies are relatively "tight", i.e., there is generally not much computation between the last write operation and the following synchronization point.

- In fanin/fanout tree operations, the individual nodes do not perform much computation, hence there is not enough work to hide communication latencies. Hence, for the communication in fanin/fanout trees, our simulations charge a one-way network latency to model the propagation of data from the source to the destination.

We are confident that despite these simplifications, our results are still useful for qualitatively comparing the relative performance of the different communication architectures. This is because all our simplifications should have a similar impact on all architectures studied. For example, on a given benchmark, the sharing pattern between nodes is the same regardless of the target architecture, hence for example the effect of changing the network's topology or routing strategy should affect all architectures in the same way.

Finally, note that the results for each of the architectures could be improved further, either by adding more hardware for latency hiding, or by using more aggressive compiler optimizations that hoist communication operations "up" from their use, thus increasing the potential for latency hiding. In that sense, our results indicate how effective these additional techniques must be for the different architectures in order to attain a given level of performance.

5.6.2 Selecting Simulation Parameters

As we mentioned before, the message-passing model already supports latency hiding through asynchronous message exchanges. The cache-coherent architecture can use many different, more specialized, mechanisms. In this section, we explore the effects of three mechanisms to reduce or hide latency in cache-coherent shared-memory machines: selectively using a write-update protocol, automatically prefetching data for sequential accesses, and asynchronously propagating write operations. For the latter technique, we assume a relaxed memory consistency model, similar to release consistency [Gharachorloo et al. 90]. Figure 5.15 summarizes the different models. Note that the study of latency reducing techniques for cache-coherent shared-memory machines is beyond the scope of this dissertation, so our list is necessarily incomplete. However, note that few existing commercial (or even research) shared-memory machines provide all the techniques we examine here.

Figure 5.16 demonstrates the effectiveness of the different latency hiding techniques; the figure shows the number of message round-trips during which processors are stalled in the different cache-coherent models. As we can see, the cache-coherent architecture benefits greatly from these relatively simple enhancements — there is a twofold to tenfold difference in latency between the best and worst models.

We have already seen that selective *write-update* improves the performance of the fanin/fanout tree operations (`broadcast`, `reduce` and synchronization) with respect to the amount of traffic generated. As we can see in Figure 5.16, write-update also drastically

Abbreviation	Description
MSG_32	Uses asynchronous writes, but reads block until data arrives.
NUMA	Uses asynchronous writes, but reads block until data arrives.
CACHE_32	Both reads and writes block. Uses a write-invalidate protocol and implements sequential consistency.
CACHE_32_wu	As CACHE_32, but can selectively use a write-update protocol where beneficial.
CACHE_32_aw	Uses asynchronous propagation of writes, and implements a relaxed memory consistency model similar to release consistency [Gharachorloo et al. 90].
CACHE_32_pre	Uses sequential prefetching of data for reads: only the first of a series of sequential cache line accesses incurs latency.

Figure 5.15: Summary of models.

lowers the *latency* of those operations. For example, for a producer/consumer sharing pattern, the write-update protocol forwards data written by the producer to the consumer in a single message. In comparison, under a write-invalidate protocol the producer would have to invalidate the consumer's copy of the cache line, and then the consumer would have to request the cache line again. Clearly, the second approach incurs much higher latency.

Asynchronous propagation of writes is especially effective in the `gauss`, `matrix` and `shallow` benchmarks, which perform many writes in a row to transfer large blocks of data. The individual write operations are effectively pipelined; the processor only has to wait at the next synchronization point for the last write to be acknowledged. This is similar to the release consistency protocol described in [Gharachorloo et al. 90]. Note that CACHE_aw still uses an invalidation-based protocol, so a consumer of newly written data still incurs the latency of requesting the data from the producer. This can be seen in the `misra` benchmark, where CACHE_aw does not significantly improve synchronization latency beyond CACHE_wu. However, when the two techniques are combined (CACHE_wu_aw), the benefits are significant.

Finally, sequential prefetching helps reduce the latency of block transfers: only the first of a sequence of sequential cache line accesses incurs latency. We see noticeable improve-

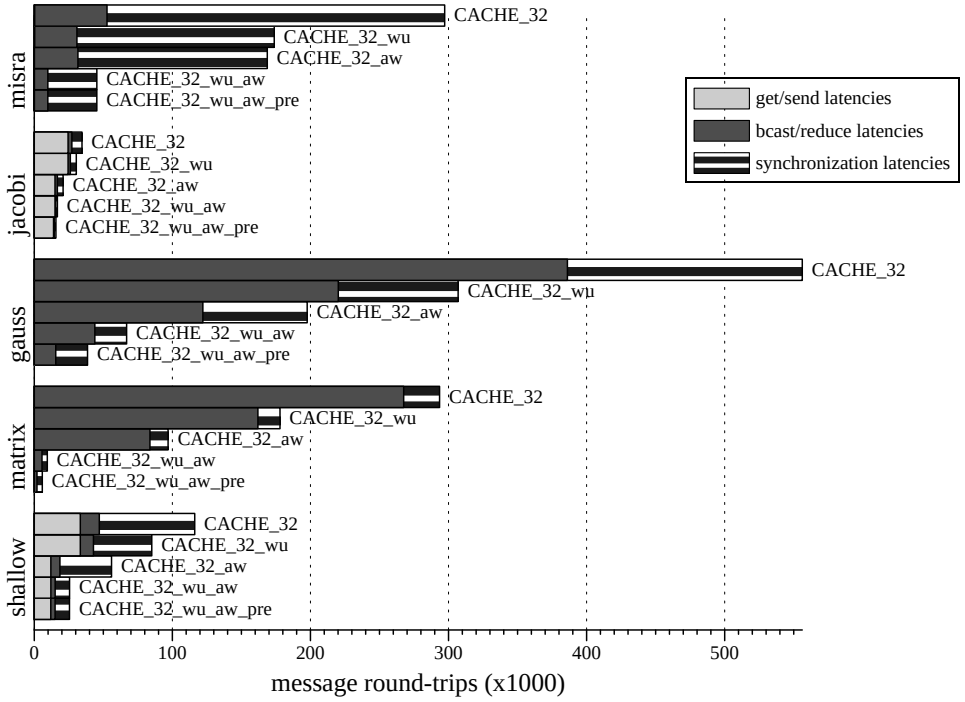


Figure 5.16: Choosing a shared-memory implementation.

ments for the `matrix` and `gauss` benchmarks, which broadcast entire rows and columns of a matrix; each row or column occupies several cache lines. The other benchmarks do not usually transfer data items occupying more than one cache line, so prefetching does not noticeably improve their performance.

We select the best of the cache-coherent models, CACHE_wu_aw_pre, for detailed examination in the rest of this chapter. For brevity, we will refer to it as "CACHE+" throughout the rest of this chapter. Note that the cache-coherent architecture could be improved further, for example, by using more sophisticated hardware prefetching mechanisms [Baer & Chen 91, Dahlgren et al. 94, Fu et al. 92], or through software-controlled prefetching [Callahan et al. 91, Klaiber & Levy 91, Mowry & Gupta 91]. Similarly, the NUMA and MSG models could be improved by adding asynchronous messages for read accesses, thus achieving an effect similar to prefetching. However, the architectural models as summarized in Figure 5.15 give us sufficient insight into the issue of communication latency.

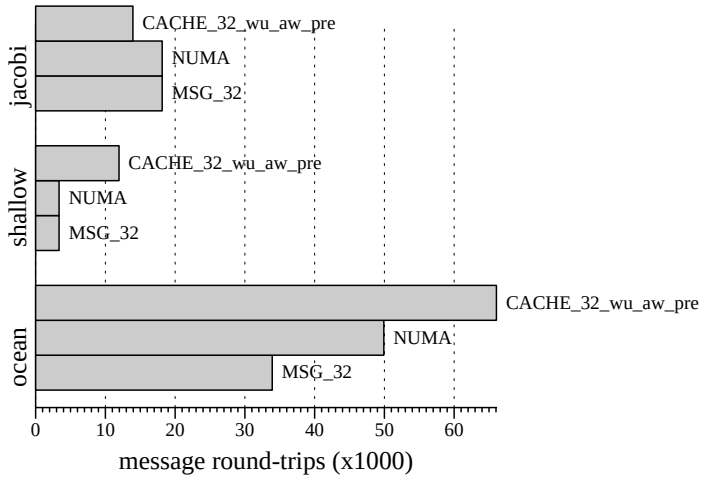


Figure 5.17: Communication latency in `get` and `send` operations.

5.6.3 Latency in `get` *and* `send` *Operations*

We now turn to evaluating the communication latency incurred by the different architectures. Figure 5.17 shows the number of network round-trip latencies incurred by the different architectures for C* `get` and `send` operations. We examine the representative `ocean`, `shallow` and `jacobi` benchmarks.

Note that despite its hardware enhancements, the CACHE+ model still incurs more latency than the other models, with the exception of the `jacobi` benchmark where CACHE+ incurs only about 75% of MSG's latency. The main reason NUMA and MSG do not perform as well as the cache-coherent model is that the former two do not prefetch data for `get` operations, whereas the CACHE+ model performs sequential prefetching.

CACHE+ incurs 380% of MSG's latency on `shallow`, and 200% of MSG's latency on `ocean`. The increase in latency is largely due to the actions of the cache coherence mechanism, which, as noted before, sometimes introduces unnecessary migrations of cache lines, which in turn results in more cache misses. Also, a fundamental difference between the cache-coherent shared-memory and message-passing architectures is that to move data from one node to another, a message-passing machine can simply send a message to the destination, whereas on a shared-memory machine, the destination node needs to *request* the data from the source node. This approach inherently incurs higher latency (a network round-trip as opposed to a one-way trip), and it is not always possible to hide the extra

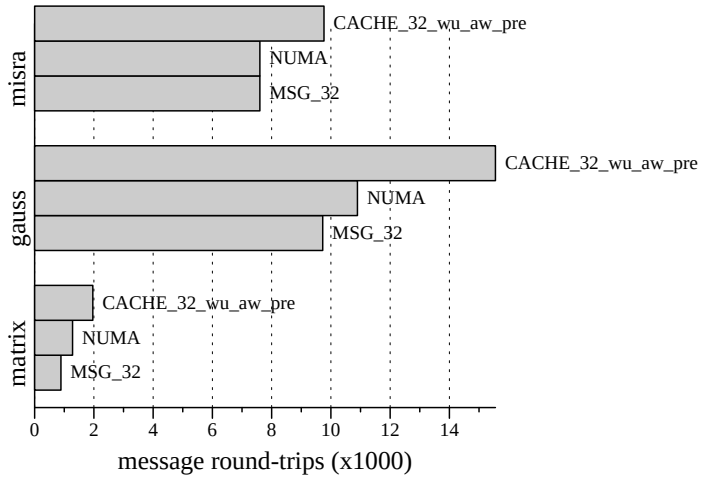


Figure 5.18: Communication latency in `broadcast` and `reduce` operations.

latency by prefetching or other techniques.

5.6.4 Latency in `broadcast` *and* `reduce` *Operations*

Figure 5.18 shows the number of network round-trip latencies incurred for `broadcast` and `reduce` operations. We examine the `matrix`, `gauss` and `misra` benchmarks which execute significant numbers of these operations.

For the `misra` benchmark, the CACHE+ model incurs only about 15% more network latency than the MSG or NUMA model, indicating that its latency hiding techniques are very effective. Its performance on the other benchmarks is not as good, however — CACHE+ incurs 50% more latency than MSG on `gauss`, and over 100% more on `matrix`. We traced most of the increase to the change in sharing patterns that occurs when one node stops broadcasting and another one takes over.[8] If the CACHE+ model gave the application more control over how and when data moves between nodes, most of this difference could be eliminated.

[8] The reason is that in the current libraries, the broadcasting node sends its data to the root of the fanout tree, by writing to a shared memory area. Normally, that memory area is shared between the broadcasting node and the root of the tree, but when a new node starts broadcasting, the memory is temporarily shared by three nodes. This interacts badly with the write-update protocol used by the fanout tree.

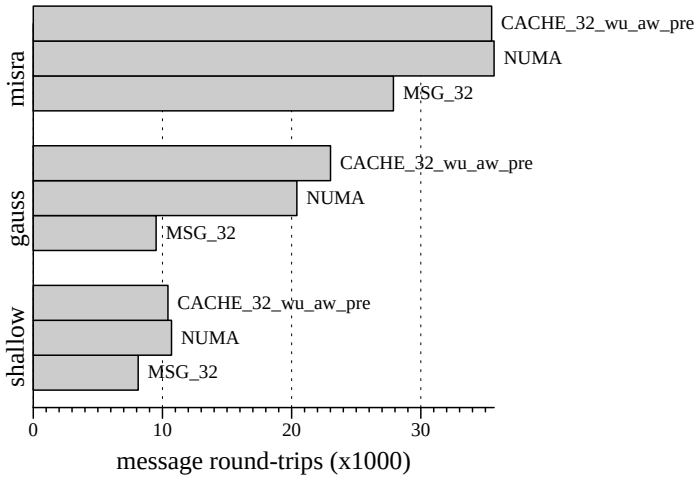


Figure 5.19: Communication latency for synchronization operations.

5.6.5 Synchronization Latency

Figure 5.19 shows the number of network round-trip latencies incurred for synchronization operations. We present results for the `shallow`, `gauss` and `misra` benchmarks; the other benchmarks have similar characteristics.

As we can see, the performance for the CACHE+ model is comparable to the performance of the NUMA model. This is not surprising, since the write-update protocol gives the cache-coherent model almost the same degree of control over data movement as the NUMA model.

At the same time, both CACHE+ and NUMA incur from 20% to over 50% more communication latency than the MSG model. The reason for this is that, as discussed before, the MSG model can sometimes combine data transfer and synchronization, hence there are fewer synchronization operations to perform in the first place.

Note that our compiler does not currently use “fuzzy” barriers [Gupta 89]. Doing so has the potential for reducing the overall synchronization latency, though we do not expect this to give either of the architectures a larger benefit than the others.

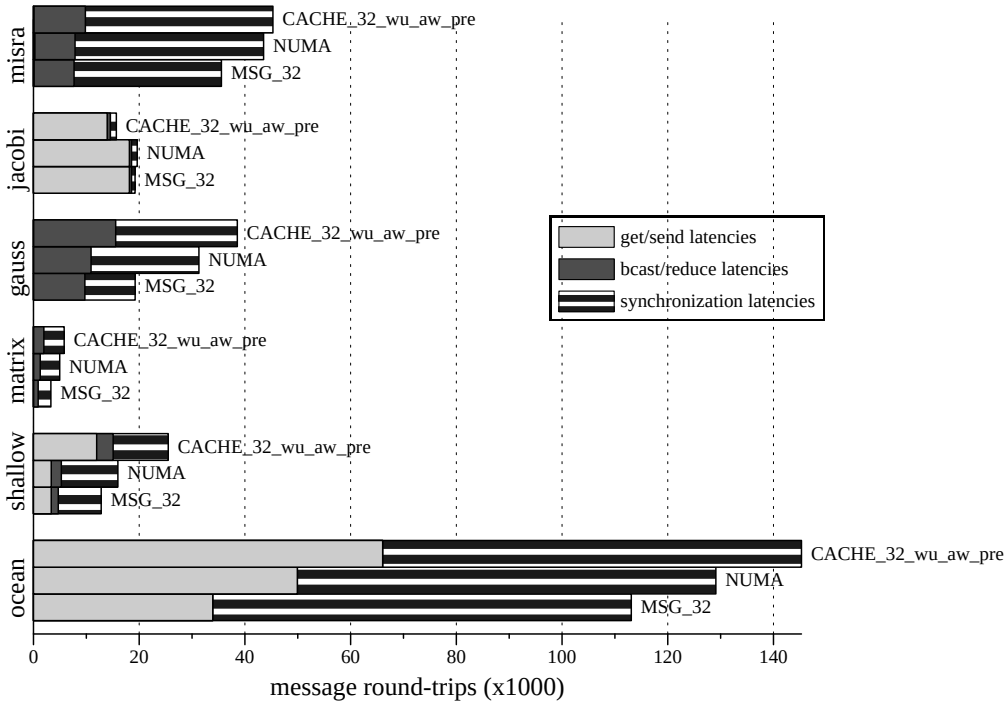


Figure 5.20: Contribution of traffic categories.

5.6.6 Contribution of Traffic Categories

To summarize the results from the previous sections, we see that the CACHE+ model generally incurs higher communication latency than the MSG model; the NUMA model's performance lies between the two. The increase can be traced to two fundamental drawbacks of the cache-coherent shared-memory architecture:

- In cases where the communication pattern is known, such as in fanin/fanout trees, the MSG and NUMA models allow the compiler to carefully coordinate the movement of data between nodes to provide the best feasible match. In the CACHE model, this is not possible — most cache coherence mechanisms are oblivious to the application's communication patterns, and even adaptive protocols [Carter et al. 91, Stenstrom et al. 93] may take some time to recognize a pattern; there is generally no way for the application to inform the hardware ahead of time of an upcoming communication pattern. Moreover, adaptive protocols have to base their decisions on observed sharing patterns, they cannot exploit à priori knowledge about the application program.

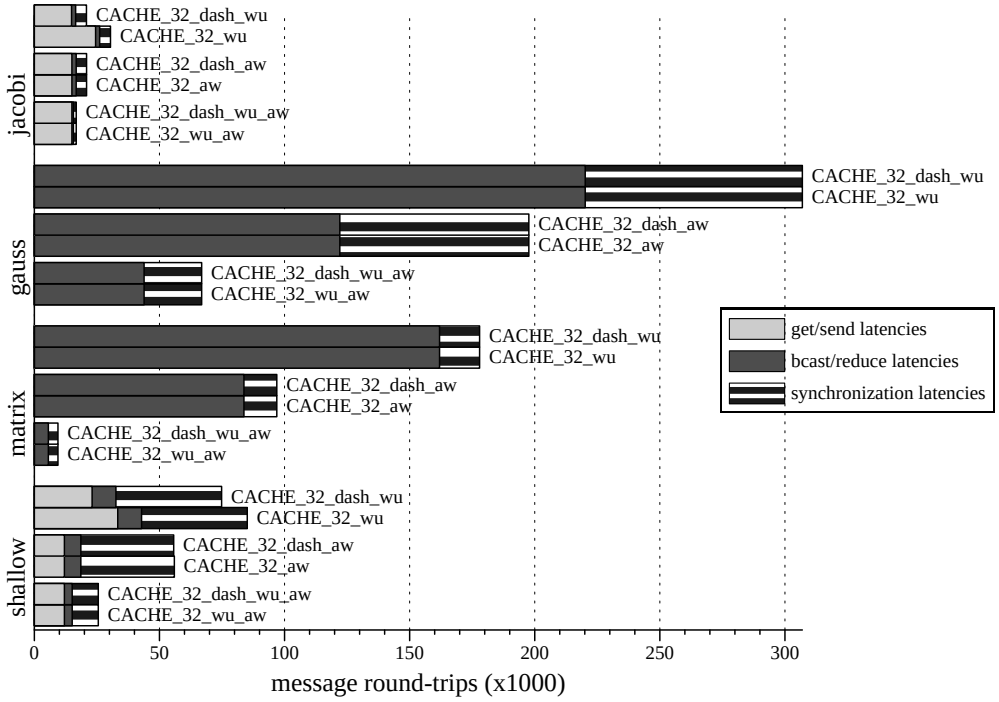


Figure 5.21: Broadcast versus point-to-point Interconnect.

- Another fundamental difference between the cache-coherent shared-memory and message-passing architectures is that to move data from one node to another, a message-passing machine can simply send a message to the destination, whereas on a shared-memory machine, the destination node needs to *request* the data from the source node. This approach inherently incurs higher latency — for example, a network round-trip as opposed to a one-way trip.

Figure 5.20 shows the latency contribution of the different C* traffic categories. Overall, the CACHE+ model, despite its many (and expensive) latency-hiding hardware enhancements described in Section 5.6.2, incurs more communication latency than the MSG model; results for the NUMA architecture lie in between CACHE+ and MSG.

5.6.7 Broadcast versus Point-to-Point Interconnect

In this section, we briefly compare the cache-coherent model based on the KSR and the one based on the DASH. Figure 5.21 shows that on some benchmarks (e.g., `shallow`), the

DASH model performs better than the KSR model. This is somewhat surprising, since remote accesses in the DASH model may require up to three network hops if the desired data is not present at the home node. However, in the KSR model all communication involves at least one network round-trip, as the KSR's network is a ring, and the coherency protocol relies on the ring's broadcast capability. On the DASH model, a given cache line's home node can, under certain circumstances, start a write operation to that cache line without having to wait for invalidations to complete. Due to the good data partitioning in our benchmarks, this case occurs relatively frequently, which can give the DASH model a slight advantage.

This is only a minor difference between the models, and it fades as more latency hiding techniques are employed. Note, however, that messages in DASH are point-to-point, whereas they must be *broadcast* in the KSR model. For a given size machine, and assuming equal technology parameters for the interconnect, one would expect message round-trip latencies to be lower in the DASH.

5.7 Related Work

Several studies have examined the performance impact of shared-memory versus message-passing *programming styles*, e.g. [Lin & Snyder 90] and [Ngo & Snyder 92]. Their experiments, performed on different shared-memory machines, show that frequently the message-passing version of a program outperforms the shared-memory version, due to better locality. Similar research comparing the performance of shared-memory and message-passing implementations of a standard cell router was performed by [Martonosi & Gupta 89]. This study, too, focused on the programming style, not on the architectural mechanisms.

In [Kranz et al. 93], the authors argue that traditional shared-memory machines suffer from the limitations of shared memory as the *only* communication mechanism available. They identify several scenarios where a compiler or programmer could implement operations more cheaply through message passing than through shared memory. By selectively using messages rather than shared memory for some communication operations, they achieved significant performance gains for runtime system primitives and one application. The results of the experiments are expressed in terms of execution time on the Alewife, and are therefore somewhat specific to that implementation, though their conclusions agree with ours.

A related approach [Frank & Vernon 93] integrates message passing and shared memory

by introducing a new cache line state, *possibly-stale*, into a conventional cache coherence protocol. The proposed architecture permits data to be moved between nodes without the overhead of cache coherence operations. At the same time, caches are kept coherent to provide a traditional shared memory model. The current studies do not yet include quantitative results.

There also is a large body of work aimed at improving the performance of the cache-coherent shared-memory architecture without changing the programming model. For example, researchers have studied adaptive or user/compiler selectable cache coherence mechanisms that use different coherency protocols for different sharing patterns [Carter et al. 91, Bennett et al. 92, Stenstrom et al. 93].

One way of combining synchronization and data transfer in the framework of a shared-memory architecture is through the use of full/empty bits on memory words [Agarwal et al. 91, Alverson et al. 90], though we would argue that this approach can be very costly, and certainly is overkill for the needs of the C* compiler.

Performance of fanin/fanout tree operations can of course be improved dramatically by providing dedicated hardware, even dedicated networks, as is done on the CM-5 [TMC 91b]. However, we know of no shared-memory machine that incorporates such hardware.

5.8 Summary

We have compiled a suite of scientific C* applications for message-passing, NUMA and cache-coherent architectures. We have simulated execution of the benchmarks and the respective architectures and measured *technology-independent* information about interconnect traffic and latency. These measurements permit evaluation of the underlying costs inherent in each of the three communication architectures. Most of our observations on traffic and communication latency can be traced back to a small number of fundamental differences between the architectures:

- Messages in a message-passing architecture carry data *and* synchronization information due to the fact that both sender and receiver can be explicitly involved in the communication operation. In contrast, messages in a NUMA or shared-memory model only carry data, therefore, those machines may have to synchronize explicitly where needed.

- In cache-coherent shared-memory architectures, data moves through the system according to a (fixed) cache coherency protocol, which is oblivious to the application's sharing pattern. For example, write-invalidate protocols perform badly on synchronization operations. Even adaptive protocols [Carter et al. 91, Stenstrom et al. 93] may take some time to recognize a pattern; there is generally no way for the application to inform the hardware ahead of time of an upcoming communication pattern. Moreover, adaptive protocols can misinterpret sharing patterns and make suboptimal decisions that a compiler could avoid.

- To move data from one node to another, the message-passing and remote-memory architectures can simply *send* the data to its destination, whereas in most cache-coherent architectures, the destination model needs to *request* the data from the source node. This approach inherently incurs higher latency (a network round-trip as opposed to a one-way trip).

- Message-passing machines can generally send data in whatever granularity is required by the application, whereas most current cache-coherent machines transfer data in units of cache lines. Especially for synchronization operations, where the "information" content of a message is theoretically a single bit, this can lead to inefficient use of the interconnect.

In particular, our experiments have shown that

- The cache-coherent models rely on spatial and temporal locality to amortize costs of the cache-coherence protocol, such as data migration or invalidation messages. Many scientific applications, especially ones employing iterative algorithms, do not exhibit much temporal locality, as all or most of the application's data set is rewritten on each iteration of the algorithm. A common example of a communication pattern that exhibits neither spatial nor temporal locality occurs when two nodes synchronize through memory operations. Again, cache-coherent models will perform badly.

- When imperfect data layout results in only part of a cache line being touched, bandwidth is wasted due to the fact that cache-coherent machines transfer data in units of entire cache lines. Unless data is rearranged dynamically (i.e., at the cost of copying), such situations cannot always be avoided.

- Message-passing architectures benefit greatly from being able to exploit synchronization implicit in the arrival of a message. Significant amounts of traffic are generated in the other models in order to effect synchronization explicitly. This is exacerbated by the above observation that synchronization operations exhibit no locality.

- The cache-coherent architecture incurs more network round-trip latencies than the other two architectures. Moreover, it requires hardware additions such as support for prefetching, selective write-update, asynchronous write propagation or relaxed memory consistency to approach performance of the message-passing architecture. The latter can use asynchronous messages as a latency hiding technique, without requiring extra hardware.

- For the benchmarks examined, the NUMA model suffers primarily from its "narrow path" to the interconnect, which often requires more control information to be sent over the interconnect. For regular traffic and tree-based algorithms, the NUMA model has an advantage over the cache-coherent model in that it affords the programmer or compiler much better control over data movement.

Although measurements of interconnect bandwidth consumed, number of messages sent and amount of network latencies incurred do not by themselves translate directly into performance figures, they are a key factor, and also point out some strengths and weaknesses of the architectures. Given technology-specific parameters such as message startup cost, bandwidth available, etc., we can derive a first approximation of actual communication cost from our measurements. Our results should therefore be considered more of a guideline for determining architectural tradeoffs rather than a direct indicator of which of the models is "best."

That said, we have demonstrated that for important regular and synchronization-intensive sharing patterns, there is a significant gap between the cache-coherent and message-passing architectures. To close this gap, cache-coherent architectures should be augmented with mechanisms that address the specific weaknesses describe above.

The Alewife machine's approach of providing both a shared-memory and a low-level message-passing interface to the interconnect may be a possible solution. Other approaches, such as user-selectable coherence protocols, fine-grain data transfer mechanisms, dedicated synchronization networks, or full/empty bits should be considered as well. As we have seen, hardware support for prefetching, asynchronous propagation of writes, or relaxed

memory consistency all help hide communication latencies. We discuss the approach of augmenting shared-memory machines in more detail in our section on future work.

However, note that most of these enhancements add significantly to the already high hardware cost and complexity of the shared-memory architecture. In comparison, the message-passing architecture achieves its advantages with minimal communication hardware.

The NUMA architecture suffers primarily from being limited to transferring at most one word per request. An efficient block transfer mechanism is essential for competitive performance (as was already noted in [Cox & Fowler 89]).

Chapter 6

IMPROVING MESSAGE-PASSING

The previous chapter has focused on the amount of interconnect traffic generated by the different architectures. In this chapter, we study a different metric, namely the CPU overhead required to send a message. Communication-related CPU overhead affects performance in two ways. First, as CPU overhead increases, so does the communication latency as seen by the application program. Second, the more time the CPU spends on communication, the less time it can spend on useful computation.

Clearly, CPU overhead is already minimal in shared-memory and NUMA architectures, since the CPU need only reference remote data using conventional `load` or `store` instructions. The NI hardware performs all the communication work and (assuming nonblocking caches) the CPU can proceed with local computations while the communication is in progress. In contrast, CPU overhead is very high in traditional message-passing machines, often over an order of magnitude higher than the interconnect latency [Felten 93b].

However, we have seen in the previous chapter that the message-passing model has many desirable features. Our goal in this chapter is to design a network interface for distributed-memory architectures that achieves low CPU overhead comparable to shared-memory machines, while at the same time retaining the advantages of the message-passing architecture that we have demonstrated in the previous chapter.

We use a *language-oriented* design approach. We first identify a small set of low-level communication and synchronization primitives that are well matched to the needs of C* (and, we argue, other data-parallel languages as well). We then design a network interface that implements these primitives efficiently and with minimal CPU overhead. Our network interface is derived from a conventional message-passing interface, and includes hardware for remote read/write requests plus counter-based synchronization support.

This chapter is organized as follows. In Section 6.1, we briefly review the sources of CPU overhead in traditional message-passing hardware and software. Section 6.2 describes the design of a traditional message-passing network interface; we show how to implement the C* communication primitives on that hardware base, and we discuss the disadvantages

of the design. In Section 6.3, we present our improved network interface design. We describe the implementation of C* communication on the new NI design, and show how the new design addresses the drawbacks of the conventional interface. In Section 6.4, we describe our experimental methodology for comparing the two designs. Our simulation results in Section 6.5 show the effectiveness of our new design at reducing per-message CPU overhead. We discuss related work in Section 6.6 and give a summary of this chapter in Section 6.7.

6.1 Problems of Traditional Message Passing

As mentioned above, traditional message-passing systems incur significant communication-related CPU overheads. A study by Felten of several scientific message-passing applications running on an iPSC/860 under NX/2 showed that these programs spend between 20% to 70% communicating, and an average of 33% of that time is communication overhead [Felten 93a]. Clearly, communication overhead can dramatically degrade the performance of parallel programs.

We distinguish two types of CPU overhead, *protocol* and *NI management* overhead. The former is a result of the semantics of message-passing; it is largely independent of the design of the NI hardware. NI management overhead encompasses all work that the CPU must do in order to interface with the NI, and hence is highly dependent on how the NI is designed. We briefly review both kinds of CPU overhead.

6.1.1 Protocol Overhead

Protocol overhead is an inevitable result of any form on inter-node communication [Felten 93a]. For example, nodes executing a parallel program must synchronize their actions in order to avoid race conditions; following [Felten 93a], we consider the associated work a form of protocol overhead. A major source of CPU overhead in traditional message-passing libraries is due to buffer management: receiving nodes must dynamically allocate buffer space for messages that arrive before the receiver has issued a matching `msg_recv` call. Since the available buffer space is finite, the nodes must execute a protocol which manages the buffers, in order to avoid deadlock. Between buffer management and other overheads due to the rich semantics of most message-passing libraries (e.g., message matching, implicit synchronization, etc.), the CPU overhead can be significant.

Felten [Felten 93a] has proposed a compiler-based approach for reducing the protocol overhead in data-parallel programs. The compiler analyzes the source program (which makes conventional message-passing calls) and extracts information about the program's communication pattern. The compiler then creates a *custom* message-passing protocol that gives the user the illusion of traditional message-passing semantics, yet reduces the protocol overhead by exploiting the program-specific information previously extracted.

As outlined in section 4.1.3, our compiler uses a different approach to reduce the protocol overhead. Instead of generating code for a message-passing communication model, it compiles code for a remote memory access model which does not require *any* buffer management. Our run-time libraries for distributed-memory architectures implement the C* communication primitives using active messages [von Eicken et al. 92], a very lightweight data transport mechanism. The residual protocol overhead exists in the form of synchronization between nodes, though as long as synchronization and data transfer can be combined in the same operation, this is a very low overhead.[1] Our approach is more ad-hoc than Felten's, yet it is very effective.

6.1.2 NI Management Overhead

Even if all protocol overhead were removed, one source of CPU overhead remains: the CPU and NI must exchange information in order to coordinate their actions. Traditionally, the NI is a passive device and, as we shall see in the next section, the CPU incurs significant overhead in managing that device. This generally involves reading and writing data and control words from and into memory-mapped NI registers, moving data between memory and the NI, fielding interrupts, etc. Since our compiler largely eliminates protocol overhead, the NI management overhead becomes more important.

In older message-passing machines, such as the Intel Delta [Intel 91b], only the operating system can access the NI. With such a design, most message-passing operations therefore incur the additional cost of a system call [Anderson et al. 91]. More modern machines, such as the CM-5 [TMC 91b], use a network interface that can be safely accessed from user-mode. We assume that all network interfaces used in our study have this property as well.

Note that unlike protocol overhead, the amount of NI management overhead is highly

[1] The amount of synchronization is guaranteed to be no more than what a shared-memory or NUMA architecture would require.

dependent on the details of the NI design. Also, software techniques alone cannot reduce the NI management overhead; we must also change the design of the NI.

6.2 Traditional Network Interface Design

To make the discussion of NI management overhead more concrete, we describe how to implement the C* communication primitives on a traditional NI. Figure 6.1 shows a traditional design for a message-passing architecture; machines like the Intel Delta [Intel 91b] or Thinking Machines CM-5 [TMC 91b] use this basic network interface design.

The NI itself consists of little more than two FIFOs (one to receive data from the network and one to hold data that is to be injected into the network) and simple control circuitry. The NI is accessible through a set of NI registers that are mapped into the CPU's address space. For example, the CPU can read successive words from the *receive* FIFO by reading one of the NI registers, or start composing a message by writing message header information into another register. Other registers may provide NI status information to the CPU, such as the space left in the *send* FIFO, or the number or words waiting in the *receive* FIFO.

The NI is a passive device; all movement of data to and from the CPU or the node's memory is initiated by the CPU. For example, to send a message, the CPU deposits the message header in one NI register and the message body in another NI register. The CPU must also check a NI status register to determine how much space is left in the *send* FIFO, and whether the last message was injected successfully. If the network imposes an upper limit on the size of a packet in the network, the CPU must also *packetize* large messages, i.e., split them into individual packets and send those one at a time.

To receive a message, the CPU extracts the message header from the *receive* FIFO, interprets the header to determine how the message is to be processed, and then retrieves the message body itself. Note that until the CPU starts emptying the *receive* FIFO, incoming data just accumulates there. To prevent traffic from backing up in the network, the NI typically interrupts the CPU when the *receive* FIFO fills up (or is about to fill up). The NI may also provide a mechanism to interrupt the CPU whenever a message with a specified tag arrives. The run-time system can use such a facility to force the CPU to receive and process a message immediately rather than waiting for the next time the *receive* FIFO fills up.

Some machines, such as the iPSC/2 [Arlauskas 88], include DMA hardware to facilitate transferring data between memory and the NI. However, the CPU must still initiate all DMA

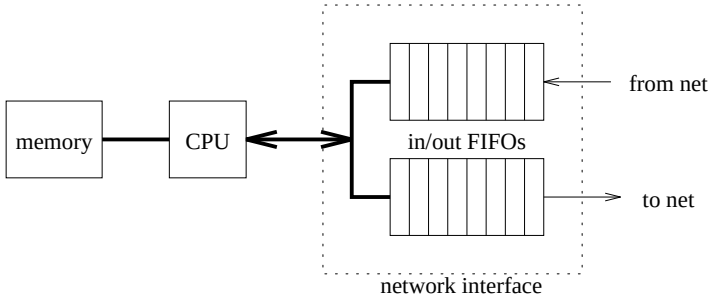


Figure 6.1: A generic message-passing network interface.

transfers. More precisely, to send a message, the CPU first creates and injects the message header, then it instructs the DMA to move the message body from memory into the NI's *send* FIFO. If the message is larger than the network's maximum packet size, the CPU also packetizes the message and initiates the DMA transfer for *each* packet. To receive a message, the CPU must at least extract enough information from the *receive* FIFO in order to start the DMA transfer, again for *each* packet. This is because messages are a very general communication mechanism; simple DMA hardware cannot decide what actions to take for a given incoming packet. Therefore, the CPU needs to actively participate in the sending and receiving of each network packet. We can see that simple DMA hardware cannot eliminate all CPU overhead; in fact, DMA hardware is not very attractive when the network's packet size is small, since the CPU overhead is proportional to the number of packets sent and received.

We now discuss how the C* communication primitives can be implemented on such an interface. Several of the primitives described below use a *fanin/fanout tree* of nodes; this tree structure is imposed *logically*, and need not be represented physically in the interconnect.

Recall that our compiler generates code for a remote memory access model of communication. Almost all communication therefore is expressed in terms of `read` and `write` messages that access memory on remote nodes.

- `send` The sending processor injects a `write` message that contains data and a destination address on the remote processor. The CPU on the receiving node extracts the message from the FIFO, examines the header and moves the data from the FIFO into memory. If the size of the message exceeds the network's maximum packet size,

the CPU must split the message into smaller packets and send each of the packets separately.

- `get` The requesting processor injects a `read` message that contains the address of the desired data and a local address where the result is to be stored. On the receiving node, the CPU extracts the message from the FIFO, examines the header, reads the memory, and injects a `write` message in reply.

- *combining* `send` The sending processor injects a message containing the data and information describing the desired combining operation. Note that the combining operation is executed by the *receiving* node; since the node processes one message at a time, this automatically serializes multiple combining operations.

- `broadcast` The node owning the data to be broadcast first sends the data to the root of the fanout tree. Then, starting at the root, each node forwards the message to its children. To reduce the overall latency of the broadcast operation, the arrival of a broadcast message generates an interrupt that causes the CPU to forward the message *immediately* instead of waiting until the next time the *receive* FIFO fills up. Our implementation of `broadcast` generates a total of p interrupts in a system of p processors. If the network supports broadcasts directly, no interrupts are needed to speed up forwarding, but each CPU must still extract the message from the NI's *receive* FIFO. Note that message-passing allows us to combine data transfer and synchronization: when a node receives data from its parent, it can also set a flag that indicates that fact to the application program.

- `reduce` Each node in the fanin-tree sends a contribution to its parent node. Once the parent has received data from all of its children, it computes the local sum and in turn sends it to its parent. When the reduction is complete, the root of the fanout tree initiates a `broadcast` of the final result. Again, the run-time library uses interrupts to minimize the overall latency of the reduction phase; a reduction requires a total of p interrupts in a system of p processors, plus another p interrupts to broadcast the result.

- `barrier` Barriers are implemented like reductions, but since barriers only occur at the end of compute phases where the CPU has no other work to do, the nodes can

poll for the barrier messages and thus avoid interrupts.

Recall that our compiler has already eliminated most of the overhead present in traditional message-passing libraries. Our run-time libraries use active messages [von Eicken et al. 92] to implement the remote read and write operations; they do not need to perform any buffer management.

With most protocol overhead eliminated, a significant amount of CPU overhead remains, namely the NI management overhead discussed earlier. As we have seen, the CPU has to actively participate in the sending and receipt of *all* messages. Specifically, it must send data and control information to the NI, read data and status information from the NI and move the data in message bodies to and from main memory. Since message receipt is inherently asynchronous, the CPU incurs an interrupt every time the *receive* FIFO fills up, or a message requiring immediate processing arrives. Alternatively, the CPU may poll for incoming messages, but this can also consume significant amounts of CPU time.

Asynchronous messages are another source of CPU overhead. We overlap asynchronous `read` and `write` messages with computation in order to hide network latencies. The run-time system must keep track of all pending asynchronous requests so it can determine when they have completed. For example, the program may transfer data from a remote node to local memory by issuing a series of asynchronous `read` requests — the local copy must not be accessed until all replies have been received. Similarly, to preserve inter-processor data dependencies, nodes may not enter a barrier until all of their messages have been delivered. Therefore, the CPU must keep a count of all outstanding asynchronous messages and also send acknowledgements for asynchronous `write` messages.

In summary, we find that the CPU performs a significant amount of work for every network packet sent and received. For example, the CPU sends data and control information to the NI, reads data and status information from the NI, moves data to and from main memory and handles interrupts from the NI. Simple DMA hardware cannot completely eliminate this overhead, since the CPU must still process the header of each network packet, and explicitly initiate the DMA transfer for each packet.

As our experiments in Section 6.5 will show, relying on the CPU to perform all of these operations can result in significant overhead. This is particularly noticeable when the compiler has already eliminated the protocol overhead of traditional message-passing libraries.

6.3 New Network Interface Design

In this section, we present a design that significantly reduces the CPU overhead for managing the NI. The key insight is that the compiler uses only a small set of well-defined communication primitives: almost all data transfer occurs through remote `read` and `write` operations and almost all synchronization involves counting messages. Our NI design is *language-oriented*, i.e., it is tuned to efficiently execute the most common operations without CPU intervention. We derive our design from a traditional message-passing interface by adding hardware support for remote `read` and `write` operations, and for message counting. The new NI retains all the advantages of the MSG model studied in the previous chapter; the only difference is that the per-message CPU overhead is much lower.

We first present the details of our design and then demonstrate how it efficiently supports the C* communication primitives. Figure 6.2 shows our language-oriented network interface design. The NI has direct access to the node's memory, and handles `read` and `write` messages in hardware. This includes sending acknowledgement messages for writes, if requested by the application. Since the application program can only specify virtual addresses in `read`/`write` messages, a translation lookaside buffer (TLB) in the NI translates and checks all of the NI's memory accesses. When a TLB miss occurs, the NI interrupts the CPU, which can then supply a valid mapping or signal an error. The TLB is entirely under software control by the CPU, and the CPU must keep it consistent with its own virtual memory maps. However, since each NI TLB only maps data local to its node, we need not keep TLBs consistent *across* nodes.

The NI contains two banks of *counters* that are used for various synchronization operations. The CPU can read and write the counters and the NI can increment or decrement them as messages are sent and received; bits in the message header can specify a counter number and a counter operation. Another bit in the header controls whether the NI should interrupt the CPU when the counter reaches zero. The *remote* counters are intended to count messages arriving from other nodes, whereas the *local* counters keep track of pending asynchronous messages originated from the local node. The compiler uses register-allocation techniques to assign counters to communication operations.

For example, to implement a communication pattern where each node receives four messages from other nodes, the compiler allocates one remote counter for the communication operation and initializes it to the value 4. All messages sent as part of the communication operation specify that the NI should decrement that counter upon receipt. When a node's

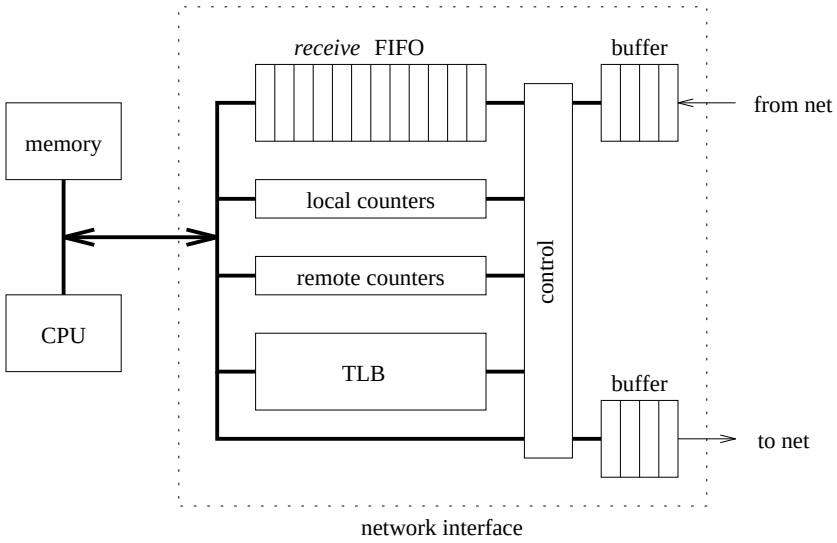


Figure 6.2: Language-oriented network interface.

counter reaches zero, this indicates that all expected messages have been received, and the communication operation is complete. Our run-time libraries use the same approach in all fanin/fanout tree operations, and our compiler could use it to combine C* data transfers with synchronization operations in order to reduce network traffic.

The local counters are designed to be used in conjunction with asynchronous messages. For example, when transferring data to a remote node using asynchronous `write` operations, the compiler allocates a local counter for the transfer. Each time the node sends a `write` message, the NI increments the counter. The NI decrements the counter when it receives an acknowledgement message from the remote node. When the counter reaches zero, this indicates that all writes have been received by the remote node.

Finally, a traditional *receive* FIFO provides message passing functionality as an escape mechanism for operations that do not fit well into a remote memory access model. Small buffers provide some amount of decoupling of the interface from the network.

C on Improved Hardware*

We now describe the implementation of the C* communication primitives on our improved hardware.

- `send` The sending CPU packetizes the `write` message, and injects the packets, including their headers, into the network. At the receiving node, the NI interprets the message header and stores the message body at the address indicated in the message header. For asynchronous operations, the message header specifies a local counter that the NI increments as the message is sent, and that it decrements when it receives an acknowledgement.

- `get` The requesting CPU injects a `read` message that contains the address of the desired data and a local address where the result is to be stored. On the receiving node, the NI receives the message, reads the requested memory area and creates and injects a `write` message in reply. The NI at the requesting node handles the `write` message. Like for `send` operations, the CPU can specify a local counter, so the CPU can detect when a series of asynchronous `get` operations has completed. We require the requesting CPU to perform packetization, i.e., it has to split large requests into separate requests such that the response fits into a single network packet. This way, the NI need not perform packetization when replying to `read` requests. We later study a version of the NI that can perform packetization as well.

- *combining* `send` To implement this operation using remote read and write operations, the run-time library would have to use *locks* to ensure that combining operations from different processors are properly serialized. Instead, we fall back on the message-passing implementation described in Section 6.2, which naturally provides serialization at the receiving node.

- `broadcast` We implement data transfer for a broadcast in terms of `write` messages; the compiler ensures that the destination of the broadcast is located at the same virtual address on each node. The compiler also allocates a remote counter and initializes it to "1"; the NI interrupts the CPU when the counter reaches zero (i.e., when the data from the parent node has arrived) and the CPU forwards the data to the children. This is only a slight improvement over the traditional NI design, but when the network has broadcast capabilities (i.e., the nodes need not forward data to their children), our design can perform the C* `broadcast` without *any* CPU intervention. With a traditional network interface, the CPU must at least extract and examine the message header, set a flag to indicate the message arrival, and initiate DMA to store the

message body in memory.

- `reduce` For a k-ary fanin tree, the compiler preallocates $k + 1$ memory locations on each non-leaf node in the fanin tree in order to hold the contributions from the node itself and the k children. The compiler also allocates a remote counter and initializes it to the value $k + 1$. Every time data arrives from a child, the NI decrements the counter. For the local contribution, the node's CPU itself decrements the counter. Once the counter reaches zero, the NI interrupts the CPU, which can then perform the local reduction step and send the result to the parent node. On the traditional hardware, each non-leaf CPU is interrupted k times, whereas our design only requires a single interrupt. In the case where the CPU makes the local contribution after all the children's contributions have arrived, the CPU can even avoid interrupts entirely.

The remote counter scheme as described above requires nodes to initialize the remote counters *before* any other node can send a message using that counter. A purely compiler-based solution must therefore initialize all counters in the preceding compute phase. However, each phase may have several control flow predecessors and/or successors, with possibly different uses of the counters in each. To prevent conflicts, the compiler would either have to use more counters or else insert additional synchronization operations. Instead, we allow the message header to specify, along with a remote counter number, an initial value for that counter. If the counter's value is still zero upon message receipt, then it is loaded with the value from the message. This way, nodes can initialize a remote counter during the phase in which it is used: all messages will include the initial value, but only the first to arrive will actually be used to initialize the counter. This technique allows us to get by with only a few remote counters — none of our simulations requires more than 16 remote counters.

We can also use remote synchronization counters to perform synchronization at the end of a compute phase. When nodes can determine *a priori* the number of messages they are to receive in a given phase, the compiler can allocate a remote counter to detect when all messages have been received. The compiler can then replace the barrier at the end of the phase with a test of the remote counter.

Network Considerations

Our decision to provide node-to-node remote memory access while allowing arbitrary numbers of outstanding asynchronous messages complicates the issue of deadlock in the network. In our design, we avoid deadlock by classifying messages into *requests* and *replies*. Replies can always be *sunk*, i.e., the NI can consume them immediately, regardless of how congested the network is or what other resources are currently in use. We assume that the network provides two separate (virtual or physical) *channels*; by using one for requests and the other for replies we prevent cycles and hence deadlock. This approach to deadlock avoidance is similar to the one used in the DASH [Lenoski et al. 92] and FLASH [Kuskin et al. 94] multiprocessors.

An interesting property of our design is that it can be readily used with a network that delivers messages out of order. Our compilation strategy guarantees that messages need only be ordered with respect to the synchronization points. Therefore, `read` and `write` messages sent within the same phase can be processed in any order.

Faster Message Injection

The CPU injects a message into the network by storing the message header and body into a set of NI control registers that are mapped into user space. For example, our NI requires three control words for a `write` message, one more than the traditional design. Both designs require additional work for packetization, for constructing the control words, and for checking whether the NI has successfully sent the message.

We can reduce the cost of message injection in several ways. First, we can add hardware to the NI that can send large messages directly out of the node's memory. The CPU only indicates the start address and size of the message, and the NI performs all memory reads and takes care of packetization. Second, the compiler can usually precompute the bit patterns for some of the control words. Even when some fields are not known at compile-time, the compiler can precompute a header *template*, such that fewer fields need to be filled in at runtime. Third, we expect programs to send sequences of messages with the same type and options; the NI can provide a way to initialize the type and option word once, and send several messages using that control word. Finally, by using extra mapping hardware as on the Cray T3D [MacDonald & Barrusio 94] or Typhoon [Reinhardt et al. 94] we can implement NUMA-style access to remote memory though conventional `load` and `store` instructions.

6.4 Experimental Methodology

We evaluate several variants of the two base architectures. The "traditional" NI design is the one shown in Figure 6.1 on page 85. In our simulations, the FIFO can receive 4 KB of messages before interrupting the CPU.[2]

model	send header	send body	recv header	recv body	packetize	broadcast
`OLD`	CPU	CPU	CPU	CPU	CPU	–
`NEW`	CPU	CPU	NI	NI	CPU	–
`OLD+bcast`	CPU	CPU	CPU	CPU	CPU	yes
`NEW+bcast`	CPU	CPU	NI	NI	CPU	yes
`OLD+bcast+dma`	CPU	NI	CPU	NI	NI	yes
`NEW+bcast+dma`	CPU	NI	NI	NI	NI	yes

Figure 6.3: Architectural models evaluated.

Implementation details of our language-oriented NI design are as follows. The TLB in the NI is fully associative with 64 entries; each entry maps a 4 KB page. The size of the *receive* FIFO is 512 bytes — since the FIFO is an escape mechanism, we expect to use it less than in the old design, so it can be smaller. Note that in this baseline model, the CPU must still perform packetization and inject the packet headers for `get` and `send` operations. For `send` operations, the CPU must also inject the message body itself.

We also evaluate several variations of the two base architectures by adding hardware support for broadcast and DMA capabilities to the NI. With a network that directly supports broadcasting, nodes need no longer forward broadcast data to their children, and the CPU does not incur an interrupt. In the designs with DMA support, we assume that the NI can send directly from the node's memory and packetize messages as required by the network. When receiving messages through the old NI the CPU must still examine *each* packet's header before it can initiate the DMA transfer. All the models studied use small (32-byte) network packets; we discuss the effect of larger packets in section 6.5.5. Table 6.3 summarizes the different architectural models; the columns indicate whether

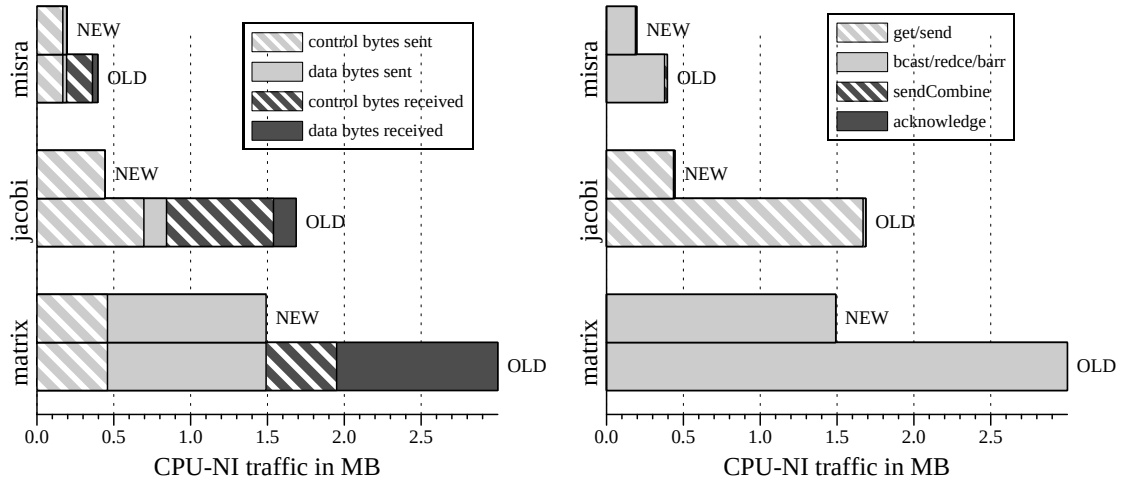
[2] By way of comparison, the *receive* FIFO on the Intel Delta is 2 KB in size.

CPU or NI/DMA are responsible for sending and receiving packet headers and bodies, and whether the interconnect provides broadcasting.

As in the previous chapter, we instrument the machine-specific communication libraries to capture the communication performed by the CPU in each of the the alternative network architectures. We gather statistics by executing the benchmarks on the KSR-1; we drive several simulation models at once to amortize the cost of program execution. For each architectural model, the instrumentation code in the run-time libraries computes the following metrics:

- Number of packets sent and received by the CPU
- Amount of data exchanged between CPU and NI
- Communication-related memory references by the CPU
- Number of communication interrupts incurred by the CPU.

These metrics summarize the CPU's communication overhead in a largely implementation-independent manner; given machine-specific timing information we can derive cycle counts from these measurements.



(a) by traffic class (b) by message type

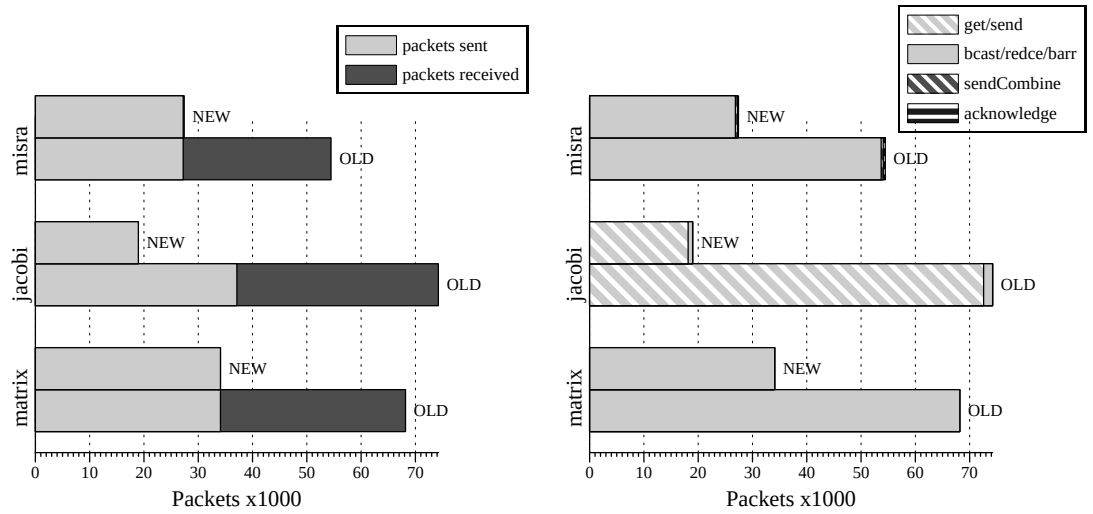
Figure 6.4: Total traffic between CPU and NI.

6.5 Results

In this section, we present the results of our simulations. As discussed in the previous section, we measure communication-related work performed by the CPU. We show graphs for the `matrix`, `jacobi` and `misra` benchmarks running on 8 nodes. Unless indicated in the text, our findings are qualitatively the same for the other benchmarks, and likewise for execution on up to 64 nodes.

6.5.1 Traffic between CPU and NI

Figure 6.4a shows the amount of data, including message headers and NI commands, exchanged between the CPU and the NI. Figure 6.4b shows a breakdown of the traffic by language-level message types. Since the new NI handles all `read` and `write` messages, the CPU need not receive and process these, as we can see in in Figure 6.4a. In all benchmarks, this effect alone reduces traffic between CPU and NI in the new design to about half that of the old design. In `jacobi`, which uses mainly `get` operations, the difference is even larger since in the new design, the NI also sends the replies to the `get` requests; in the old design this must be done by the CPU. On average, our improved design reduces traffic between CPU and NI by 50% to 75% compared to the old NI design.



(a) by traffic class (b) by message type

Figure 6.5: Number of packets sent and received by CPU.

Figure 6.5a shows the number of packets the CPU receives from or injects into the NI. This is an important metric, since the CPU incurs per-packet startup costs, such as checking for successful message injection or dispatching on message type for message receipt. The results here parallel those for message traffic — our improved design consistently outperforms the traditional hardware by a factor of two to three.

6.5.2 Communication-related Memory Accesses

Figure 6.6 shows the CPU's communication-related memory traffic. In `jacobi`, almost all messages are `read` requests and `write` replies for `get` operations. Since the new design handles these in the NI, the CPU does not need to touch memory at all. In the old design, the CPUs are responsible both for creating and receiving the reply messages. Therefore, while the CPUs in the old model must access about 300 KB of memory, virtually no communication-related memory traffic is required in our model. (The residual traffic is due to `reduce` operations.)

For `matrix`, both NI designs need to forward broadcast messages. However, in our design the incoming data is stored in memory by the NI, whereas in the old design this is the CPU's responsibility. We later show how these results change when the network directly

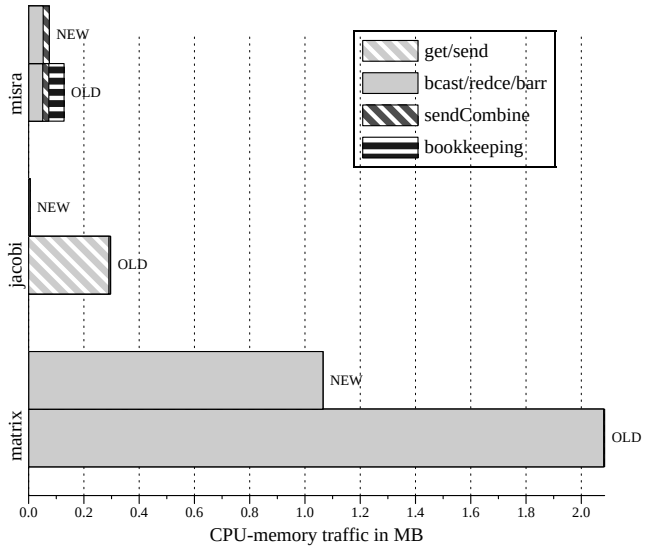


Figure 6.6: CPU memory accesses by message type.

supports broadcasting.

The `misra` benchmark shows a different source of overhead. Almost all messages are part of reduction or barrier operations (see Figure 6.5b). Very little data is carried by these messages, but nodes must count the number of contributions they have received from child nodes. In our design, the compiler allocates a remote counter and the NI does the counting. In the old design, the CPU must update a set of counters in memory. This results in a significant overhead, shown under the category of "bookkeeping" in Figure 6.6. Note that combining `send` messages are implemented the same way on both models, hence there is no difference in memory traffic.

Overall, we find that in our design the CPU performs significantly fewer communication-related memory accesses than with a traditional NI design — usually about 50% of the memory traffic of the old design, and as little as 2% for `jacobi`.

6.5.3 Communication-related Interrupts

Figure 6.7 shows the number of communication-related interrupts incurred by the CPU, broken down by the reason for the interrupt. A *synchronization* interrupt occurs when the sender of a message requests an interrupt upon message delivery, or when a synchronization counter reaches zero. Recall that our run-time libraries generate interrupts for `broadcast`

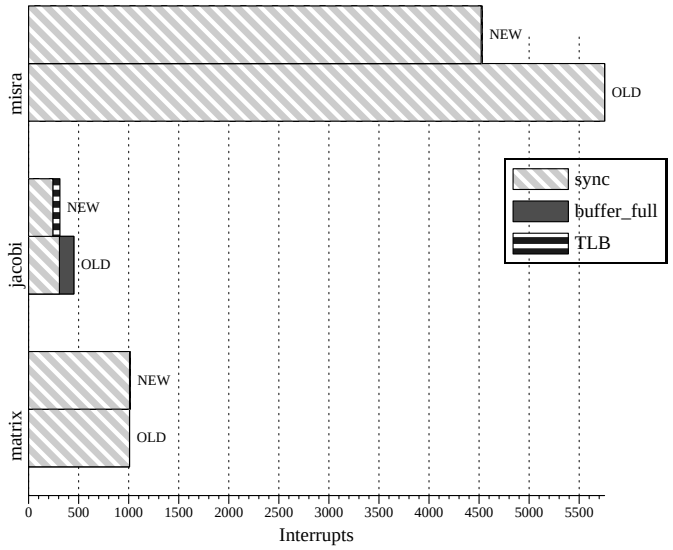


Figure 6.7: Number of CPU interrupts, by reason.

and `reduce` operations, to reduce the overall latency of the operation. When a memory access by the NI misses in the TLB, the NI generates a *TLB* interrupt. Finally, when the *receive* FIFO fills up, the NI sends a *buffer-full* interrupt to the CPU.

As mentioned before, all communication in `matrix` is due to broadcasts. To keep the latency of broadcasts low, our runtime system ensures that a broadcast message generates an interrupt on the receiver, such that the receiving node can forward the message to its children immediately — as opposed to at the end of the current compute phase, or the next time the *receive* FIFO fills up. Our synchronization counters cannot improve performance here; only direct hardware support for broadcasts eliminates these interrupts.

The `jacobi` benchmark shows again the advantage of supporting remote read and write in hardware. In the old NI, all communication flows through the *receive* FIFO, which causes an interrupt whenever the FIFO fills up. This does not happen with the new design, which stores incoming data directly into memory. `jacobi` also uses reductions, which can take advantage of the remote synchronization counters, as discussed below. The new NI incurs a noticeable number of TLB interrupts, but all of them are due to *cold* misses, and their number does not increase for longer running times of the benchmark. If the NI's TLB were warm-started, these interrupts would be eliminated. Alternatively, doubling the size of the pages mapped by the TLB halves the number of cold misses while also increasing

the maximum amount of memory the TLB can map at once. Including the TLB interrupts, the new NI generates about 25% fewer interrupts than the old NI, or 50% fewer interrupts than the old NI if we do not count cold TLB misses.

Both `misra` and `jacobi` make frequent use of `reduce` operations. As for broadcasts, the run-time libraries use interrupts to reduce the overall latency of the reduction operation. With the old NI, each message from a child to its parent in the reduction tree causes an interrupt at the parent. This allows the parent to count the contributions and start the local reduction as soon as all children have sent their contribution. Our new NI uses the remote synchronization counters to count the children's contributions and only interrupts the CPU when *all* contributions have arrived. Note that in our simulations, we use binary reduction trees; our technique is much more efficient for higher-arity trees. Also, in some cases we could avoid the interrupt entirely, though we do not exploit this feature in the simulations. The results in the figure therefore show the worst-cast for the improved design. Even under these conservative assumptions, the use of synchronization counters can reduce the number of interrupts in reductions or broadcasts by up to 20% compared to the traditional network interface.

6.5.4 Broadcast and DMA Capabilities

We now present the results for variants of the baseline models that have hardware support for broadcast and augmented DMA capabilities. Refer back to Figure 6.3 on page 93 for a quick summary of the different models.

When the network supports broadcast, the nodes do not have to forward data to their children any more, and the NI does not need to interrupt the CPU. This reduces the number of messages the CPU has to inject, and the number of interrupts it has to handle. Note that the data being broadcast must still be stored in the receiving nodes' memory.

We also add DMA capabilities to the old design, and augment our new design's capabilities to match. To send a `write` message, the CPU need only pass to the NI the message header and the start address and size of the data to be sent. The NI reads, packetizes and sends the data without further intervention by the CPU.

There remains one crucial difference between the old and new NI designs: when receiving a message, the new NI design interprets the message header and handles `read` and `write` messages without CPU intervention. With the old design the CPU needs to interpret the message header and initiate the DMA transfer. This is because the old NI itself

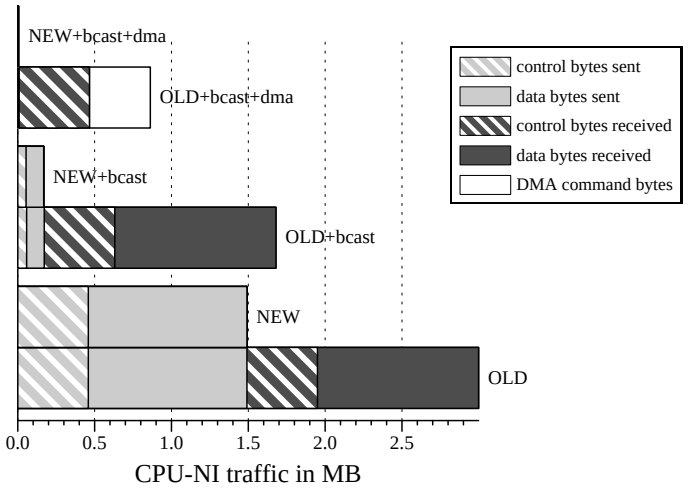


Figure 6.8: CPU–NI traffic with broadcast and DMA, `matrix` benchmark.

does not handle `read` and `write` messages directly. The impact of this minor difference is significant: recall that nodes receive large messages one packet at a time and the CPU on the receiving node must read and examine the header and initiate DMA transfer for *every* packet.

Figure 6.8 shows the control and data traffic between CPU and NI for the different variations of the old and new NI. A new traffic category, "DMA commands," represents the control information the CPU sends to the NI in order to initiate DMA transfers. Only the old NI design generates such traffic. The `matrix` benchmark broadcasts large blocks of data; the results for `gauss` are similar since it shares the same characteristics.

The basic new design generates about half as much traffic as the old design. Adding a broadcast-capable interconnect reduces the traffic for both designs, since the nodes need no longer forward broadcast data to their children in the fanout tree. We also see that the relative difference between the two designs has *increased*. In the new design, the NI on each node deposits the incoming broadcast data directly into memory without CPU intervention. In the old design, the CPUs must still receive the data and store it in memory; the figure shows that hardware broadcast reduces the number of bytes sent by the CPU, but not the number of bytes received.

With DMA support the CPUs needs no longer send data to the NI. However, with the old NI, the CPU must still examine the header of every network packet it receives, and

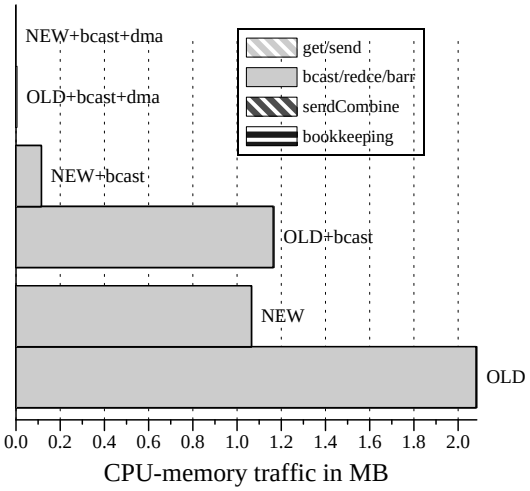
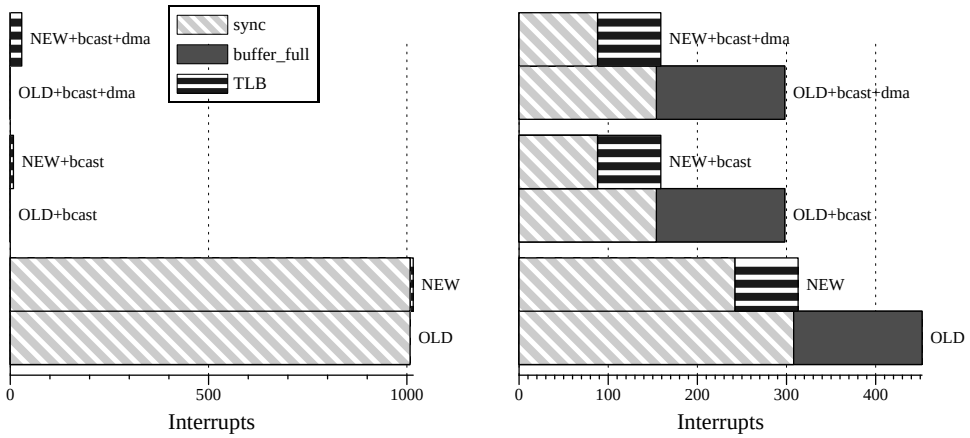


Figure 6.9: Memory traffic with broadcast and DMA, `matrix` benchmark.

initiate the DMA transfers. In the new design, this is handled entirely by the NI. The figure shows that in the new design with broadcast and DMA, the CPU hardly sends any data at all. In contrast, with the old design the CPUs receive about 500 KB of message headers and send over 300 KB of commands to the DMA hardware. Note that this overhead is particularly noticeable with the small packet size we have chosen. As we will see later in Figure 6.11, larger packets alleviate this problem somewhat.

Figure 6.9 shows the CPU's communication-related memory traffic. As we can see, the augmented designs perform substantially better than the baseline models. Note the performance difference between the old and new NI designs when broadcast but not DMA support is added — with new NI, the CPU accesses 100 KB of memory compared to 1.1 MB with the old NI. The reason for this is that the new NI handles the receipt of the broadcast message, whereas with the old NI, every node must receive the broadcast data through the CPU. Both models perform equally well when *both* broadcast and DMA support are available.

Results for the `gauss` and `shallow` benchmarks resemble those of the `matrix` benchmark, and we have seen some performance improvement for the `misra` benchmark. For `jacobi`, neither hardware broadcast nor DMA support reduces traffic between the CPU and memory or the NI. This is because `jacobi` transfers data at a granularity of 8 bytes per message, which makes DMA unattractive. We later discuss a version of `jacobi` that takes



(a) `matrix` benchmark (b) `jacobi` benchmark

Figure 6.10: Interrupts with broadcast and DMA.

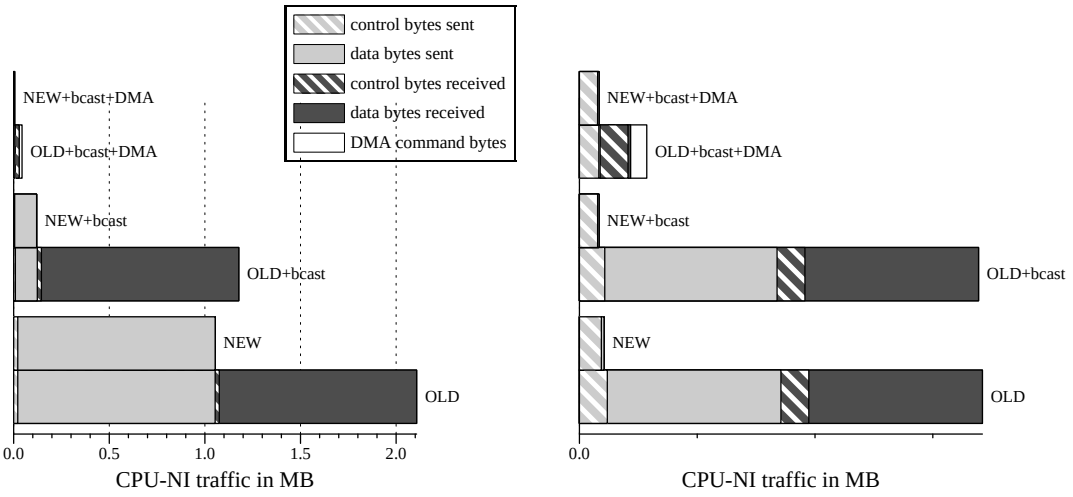
advantage of large messages.

Figure 6.10 shows how adding broadcast and DMA support affects the number of communication-related CPU interrupts. In `matrix`, shown in Figure 6.10a, almost all interrupts are caused by `broadcast` operations; implementing broadcasting in the network completely eliminates these interrupts, with both the new and old NI. Note also that with DMA (i.e. sending out of memory), the new NI performs a larger fraction of the memory accesses, which explains why the number of TLB misses increases slightly. On the other hand, all TLB interrupts shown in the figure are caused by cold misses. Warm-starting the NI's TLB would eliminate all TLB interrupts.

In `jacobi`, shown in Figure 6.10b, broadcasts are used infrequently, to distribute the result of reductions. As for `matrix`, adding direct broadcast support eliminates the interrupts for that operation, but adding DMA has no effect. The remaining "synchronization" interrupts are caused by reduction operations.

6.5.5 Large Packet Sizes

We also simulate our benchmarks for a network that supports arbitrarily large messages. For this experiment, we modified the `jacobi` benchmark to aggregate the small per-VP requests into a single large message. This is easy for a compiler, since the communication pattern in `jacobi` is static and very regular. Figure 6.11 shows the traffic between CPU



(a) `matrix` benchmark (b) `jacobi` benchmark

Figure 6.11: Total traffic between CPU and NI, large packets.

and NI for a network using large packets. As expected from larger packets, the ratio of data payload to packet header (control) information increases drastically, except in `misra` (not shown) which simply cannot exploit large packets.

Comparing the results in Figure 6.11 to those obtained on a network with small packets (Figure 6.4 on page 95), we find a significant reduction in traffic, for both the old and new NI designs.

However, the relative difference between the old and new designs is now much higher than it was for small packets. We conclude that while larger packets improve absolute performance of all NI designs, the new design can take better advantage of the larger packets.

6.6 Related Work

Several researchers have proposed software techniques to reduce message-passing overhead. Active messages [von Eicken et al. 92] are a low-level transport mechanism that achieves low latency by efficiently dispatching to a message handler on the receiving node. Felten [Felten 93a] proposes using a *protocol compiler* to custom-generate message-passing protocols for a given program and thus reduce protocol overhead. Neither approach can

completely overcome inadequacies of existing network interface hardware.

The Intel Paragon [Intel 91a] uses a second i860 microprocessor on each node to handle communication operations. However, the network interface is not accessible at the user level and all communication must pass through the coprocessor. This requires synchronization between the two processors, which for small messages may be more costly than allowing either processor to directly access the network interface, as is done on the CM-5 [TMC 91b]. Given that the coprocessor is identical to the compute processor, it may be more efficient to use it for general computation, performing communication through the kernel on both processors.

The Shrimp architecture [Blumrich et al. 94] implements a low-overhead data transport mechanism by marking memory pages as "mapped out"; `store` operations to those pages cause the written data to be automatically forwarded to the memory of another node. The mapping, established by the OS kernel, specifies destination node and address. Our work focuses more specifically on the needs of a parallel compiler, and on the separation of data transport and synchronization.

[Thekkath et al. 93] propose a remote memory access model instead of traditional message-passing for streamlining data and control transfer between workstations on a local area network. Though our approach is similar in nature, our emphasis lies on user-level communication and compiler support for parallel programming; their emphasis is on distributed applications.

In [Henry & Joerg 92b], the authors propose a network interface design that provides special support for Id [Nikhil 90] programs that have been compiled to Berkeley's Threaded Abstract Machine [Culler et al. 91a]. They reduce communication overhead by implementing message dispatching, forwarding and replying in hardware, and by mapping the network interface into the processor's general-purpose registers. Compared to our work, they target a much more fine-grained computation model, yet reach similar conclusions.

The FLASH multiprocessor [Kuskin et al. 94] and Typhoon [Reinhardt et al. 94] use very flexible network interfaces built around a fully programmable microprocessor core. While the aim of these architectures is to support shared-memory systems, it should be possible to implement our communication primitives on their network interfaces. They would therefore provide an ideal testbed for the communication architecture proposed in this chapter.

6.7 Summary

In the previous chapter, we have seen that message passing architectures do have several inherent advantages. However, the software cost of sending and receiving data in a message-passing machine is generally high, negating any possible performance advantages [Felten 93b].

Our goal is to reduce the overhead of data transfer and synchronization in message-passing distributed-memory architectures. We leverage off of the assumption that in the future, fewer programmers will use message passing primitives directly; instead, programs will be written in high-level parallel languages or generated from sequential code using parallelizing compilers. By using a high-level parallel compiler, it is possible to exploit information about communication patterns and perform at compile-time many of the tasks traditionally provided by a communication library. Specifically, the compiler can eliminate most of the protocol overhead of traditional message-passing libraries [Felten 93a].

However, the communication primitives offered by traditional message-passing network interfaces do not match well the needs of compilers. We have found that implementations of C* on traditional message-passing hardware require significant CPU overhead for communication. Specifically, since those network interfaces are typically passive, the CPU must participate in the receipt of all messages; all communication traffic flows through the CPU. As message receipt is inherently asynchronous, the CPU must either poll for incoming messages, or incur interrupts.

We observe that in our benchmarks, nearly all data transfer is done by reading and writing remote memories. Similarly, nearly all synchronization is performed by counting messages sent or received. Both of these operations are simple enough to be provided by the hardware, and the compiler can use them as building blocks to implement C* communication operations.

Based on these observations, we have proposed an improved network interface design that supports remote read and write operations in hardware, and provides a set of *synchronization counters* that the NI manipulates as part of message handling. The user is given full control over how counters are used by the NI and the compiler can thus combine data transfer and synchronization where appropriate. The NI counters can also be used to provide efficient support for asynchronous messages, reducing the amount of bookkeeping the CPU must do.

Compiler support is essential in order to best make use of our design. For example,

register-allocation techniques are used to choose synchronization counters. The compiler must also infer the destination addresses for remote memory accesses, allocate buffer space, and insert synchronization operations in order to preserve inter-node data dependencies.

To compare our language-oriented design against traditional network interfaces, we simulated the execution of a set of C* benchmarks on both architectures. We measured the traffic between the CPU and the NI, the amount of communication-related memory traffic, and the number of interrupts incurred by the CPU. For our C* implementation, these measurements capture the dominant sources of CPU overhead in a timing- and technology-independent manner.

Our results demonstrate the effectiveness of our design at reducing communication-related CPU overhead. Traffic between CPU and NI is reduced by at least 50%, and even as much as 90% in some cases. We find similar reductions in traffic between CPU and memory. For some benchmarks, our design also achieves a substantial reduction in the number of communication-related interrupts.

We conclude that as high-level parallel languages become more common and fewer programs directly use traditional message passing primitives, *integrated* compiler and hardware design approaches are essential for achieving good communication and synchronization performance.

Chapter 7

CONCLUSIONS

The work described in this dissertation is motivated by a growing trend towards using high-level parallel languages to program parallel computers. To date, existing communication architectures, such as message passing, remote-memory access and shared memory, have been primarily used (and designed to be used) directly by the programmer. This bias has influenced their design, much as assembly language programming has influenced the design of CISC instruction sets. For example, one commonly cited argument in favor of shared-memory machines is that they are easier to use than message-passing — a statement that clearly reveals a design bias towards simplifying the communication architecture for the benefit of the programmer.

In the future, programs will be compiled to the specific parallel target; the compiler will hide the details of the underlying communication architecture from the programmer. Hence, the programmer's convenience is no longer a major concern in the design of the communication architecture, since the programming language already provides a convenient programming model. Instead, performance becomes a driving concern, and the communication architecture must provide interfaces that best suit the needs of the compiler. This approach is similar to the RISC philosophy in processor design.

In this dissertation, we have focused on the class of *data-parallel languages* and have picked the C* language as one representative for our experiments. We have compared three communication architectures — message-passing, remote-memory access and shared-memory — for a set of scientific benchmarks written in C* and compiled to the respective architectures.

A fair evaluation of such fundamentally different architectures has so far been difficult to produce for several reasons. First, programs were typically hand tailored for different architectures, often resulting in vastly different algorithms for the same application. Second, while program execution time can be measured on different multiprocessors, such measurements are difficult to compare, since the many *implementation* differences between machines — such as processor architecture and cycle time, memory system details, and

bus technology — tend to obscure the *architecture-inherent* differences in which we are interested.

We avoid the first problem by using a *single* suite of benchmarks written in C*, which are compiled to the architectures under consideration. We thus measure the work required by each architecture to execute the *same* data-parallel programs.

To address the second problem, our simulations abstract implementation details and instead focus on metrics that are not affected by implementation details like processor or network speed. We have measured the number of messages sent, the total interconnect control and data traffic, and the number of round-trip communication latencies incurred. We have deliberately rejected overall execution time as a metric, since it cannot yield the same kind of fundamental insight as our implementation-independent metrics.

The drawback of our approach is that our results do not by themselves translate into absolute performance; we would need to know machine-specific data such as processor speed, time to send a message, network latency, etc., in order to derive a first-order estimate of execution time. On the other hand, our method can point out differences that are inherent in the architectures, rather than any specific implementations of the architectures.

Our results have shown that message-passing has several important advantages over the competing architectures. The shared-memory model sends between 6% to 500% more messages than the message-passing model and requires 25% to 250% more bandwidth. Even with aggressive hardware support for latency hiding, the shared-memory model incurs from 20% to 100% more network round-trip latencies than the message-passing model on all benchmarks except `jacobi`, where it incurs 15% *fewer* round-trip latencies than the message-passing model.

We have identified three architectural differences that account for these results. First, in a message-passing model, the compiler has full control over when data is moved between nodes and at what granularity. In shared-memory, data movement is largely under the control of the cache-coherence mechanism, and usually occurs at a granularity of entire cache lines. Second, message-passing allows the compiler to combine data transfer and synchronization in a single message, whereas two separate operations are required in a shared-memory model. Third, to move data from one node to another, the message-passing and remote-memory architectures can simply *send* the data to its destination, whereas in most cache-coherent architectures, the destination model needs to *request* the data from the source node, an approach that inherently requires more trips through the network and hence

incurs higher latency.

However, traditional message-passing implementations also have a major drawback: the amount of work the CPU must perform for each message (the *communication overhead*) is very high, which hurts performance — especially when programs send many small messages. The shared-memory and remote-memory access architectures do not have this problem. An ideal communication architecture for C* would unite the above advantages of the message-passing architecture with the low communication overhead of the shared-memory architecture.

As one possible solution, we have proposed a *language-oriented* design that retains the advantages of the message-passing model, yet in cooperation with the compiler significantly reduces the per-message overhead. To do so, we have identified a small set of low-level communication and synchronization primitives that are well matched to the needs of C* and then designed a network interface that efficiently supports these primitives.

Our network interface includes hardware for remote read/write operations plus counter-based synchronization support. These primitives are a good match for C* (as well as similar data-parallel languages, such as HPF), since almost all communication operations in C* read or write variables (i.e., memory) on remote nodes. Similarly, the communication libraries can easily perform inter-node synchronization by counting messages.

To evaluate the effectiveness of our approach, we have simulated and measured our compiled C* benchmarks on a traditional message-passing interface as well as our language-oriented design. These measurements have demonstrated that our design is effective at reducing communication-related CPU overhead. Compared to a traditional message-passing NI design, the CPUs in our improved design exchange 50% to 75% less data with our NI, perform 50% to 90% fewer communication-related memory accesses on average, and incur up to 20% fewer interrupts for broadcast and reduction operations. Our design is also better able to exploit networks with broadcast capabilities.

7.1 Future Work

Our work could be extended in several ways. First, we could improve the compiler, especially for the message-passing architecture. Second, in this dissertation we have proposed a language-oriented design derived from message passing; an alternative would be to design a network interface based on a shared-memory model. Finally, we could extend our study to programming models that are not data-parallel.

7.1.1 Improved Compiler

Though the message-passing model already compares favorably to the other architectures, we could significantly improve its performance by using more aggressive communication optimizations. For example, our compiler does not currently perform message vectorization or message aggregation.

Our compiler also does not fully exploit the capabilities of our new NI design; we only use the synchronization counters for synchronization internal to `broadcast`, `reduce` or `barrier` operations. For regular communication patterns, the compiler could precompute the number of messages expected by each node and use the counters to synchronize, instead of a barrier operation.

We could also improve the shared memory model's performance on `jacobi` by copying non-contiguous data into a contiguous memory area before the neighboring node accesses it; this would improve the utilization of the cache lines. Note that this optimization presumes a fairly static and regular communication pattern that is amenable to detailed analysis by the compiler. It is likely that the same compile-time analysis would yield an even more efficient implementation on a message-passing machine, where the compiler has better control over data movement.

7.1.2 Enhancing Shared-Memory Architectures

In this dissertation, we have proposed a language-oriented network interface design that was derived from a traditional message-passing network interface. The goal was to attack the drawback of traditional message-passing, namely the high communication overhead, while retaining the advantages of the message-passing model.

A different approach would be to start with a shared-memory architecture and modify the design to address its specific shortcomings, e.g., by giving the compiler better control over data movement, or providing communication mechanisms that can combine data transfer and synchronization in a single operation. We have already mentioned enhancements such as prefetching, compiler-selected coherence protocols, asynchronous write propagation, relaxed consistency models or full/empty bits. Machines like FLASH [Kuskin et al. 94] or Typhoon [Reinhardt et al. 94] would be ideal testbeds for such an approach, since their network interface is fully programmable.

Another promising approach is to build hybrid architectures that support both shared-memory and message-passing, such as the Alewife [Agarwal et al. 91]. The challenge here

is to carefully integrate the different communication models in the compiler.

The guiding force behind any such effort should the the principle that whenever the compiler can identify a particularly efficient approach to communication, the hardware must not stand in the way or hide too many details of the communication architectures.

7.1.3 Extending to Wider Class of Programs

Our work has so far focused on data-parallel high-level programming languages. While this is an important class of languages, not all algorithms can be expressed efficiently in a data-parallel form. It would be useful to examine how the communication requirements for other programming models differ from those of data-parallel languages. For example, Henry and Joerg designed a NI for use with the TAM [Culler et al. 91b] model of execution and reached conclusions that are somewhat different from ours [Henry & Joerg 92a]. While this is a fairly extreme example, in that their programming model is radically different from the data-parallel model, it shows the tight connection between the choice of a programming model and the design of the communication architecture.

7.2 Conclusions

We have identified architecture-inherent advantages and disadvantages of the message-passing, remote-memory access and shared-memory communication architectures, and have shown a way to remedy the deficiencies of the message-passing model. Thus, the reports of the demise of message passing have been greatly exaggerated. We believe that the shared-memory model is also amenable to enhancements that address its shortcomings. Future work may tell which of the two architectures, message-passing or shared-memory, provides the better starting point for high-performance, low-overhead communication architectures.

Bibliography

[Adve & Hill 90] S. Adve and M. Hill. Weak ordering — a new definition. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 2–14, 1990.

[Agarwal et al. 88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of 15th International Symposium on Computer Architecture*, pages 280–289, 1988.

[Agarwal et al. 91] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[Alverson et al. 90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of International Conference on Supercomputing*, pages 1–6, 1990.

[Anderson et al. 91] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. Performance measurements on a 128-node Butterfly parallel processor. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[Archibald 88] J. K. Archibald. A cache coherence approach for large multiprocessor systems. In *1988 International Conference on Supercomputing*, pages 337–345, 1988.

[Arlauskas 88] R. Arlauskas. *iPSC/2 System: A Second Generation Hypercube*, January 1988.

[Baer & Chen 91] J. L. Baer and T. F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings Supercomputing '91*, pages 176–186, 1991.

[Barrusio 94] R. Barrusio, 1994. Usenet Communication on `comp.sys.super`.

[Bennett et al. 90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 125–134, 1990.

[Bennett et al. 92] J. K. Bennett, S. Dwarkadas, J. Greenwood, and E. Speight. Willow: a scalable shared memory multiprocessor. In *Proceedings. Supercomputing '92*, pages 336–345, November 1992.

[Blumrich et al. 94] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. To appear in *Proceedings of 1994 International Symposium on Computer Architecture*, 1994.

[Bokhari 90] S. Bokhari. Communication overhead on the intel iPSC-860 hypercube. Technical Report Interim Report 10, ICASE, May 1990.

[Brustoloni & Bershad 92] J. C. Brustoloni and B. N. Bershad. Simple protocol processing for high-bandwidth low-latency networking. Technical Report CMU-CS-93-132, School of Computer Science, Carnegie Mellon University, March 1992.

[Callahan et al. 91] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[Carter et al. 91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Censier & Feautrier 78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, pages 1112–1118, December 1978.

[Chapman et al. 92] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran — a Fortran language extension for distributed memory systems. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-time Environments for Distributed Memory Machines*. Elsevier Press, 1992.

[Clark et al. 92] T. W. Clark, R. von Hanxleden, K. Kennedy, C. Koelbel, and L. Scott. Evaluating parallel languages for molecular dynamics computations. In *Proceedings. Scalable High Performance Computing Conference SHPCC-92*, pages 98–105, April 1992.

[Cox & Fowler 89] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor. experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[Cray 93] Cray Research, Inc., 2360 Pilot Knob Road, Mendota Heights, MN 55120. *CRAY T3D System Architecture Overview Manual (HR-04033)*, 1993.

[Crowther et al. 85] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance measurements on a 128-node Butterfly parallel processor. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 531–540, 1985.

[Culler et al. 91a] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine grain parallelism with minimal hardware support: A compiler-controlled treaded abstract machine. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[Culler et al. 91b] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled

threaded abstract machine. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, April 1991.

[Cypher et al. 93] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of 20th International Symposium on Computer Architecture*, pages 2–13, 1993.

[Dahlgren et al. 94] F. Dahlgren, M. Dubois, and P. Stenstrom. Combined performance gains of simple cache protocol extensios. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 187–197, 1994.

[Dally 90] W. J. Dally. The J-machine system. In P. Winston and S. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, volume 1. MIT Press, 1990.

[Eggers & Jeremiassen 91] S. Eggers and T. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:377–381, August 1991.

[Felten 93a] E. W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. PhD dissertation, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, September 1993. Available as technical report 93-09-09.

[Felten 93b] E. W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. PhD dissertation, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, September 1993. Available as technical report 93-09-09.

[Felten 94] E. W. Felten, 1994. Personal Communication.

[Fox 88] G. C. Fox. What have we learnt from using real parallel machines to solve real problems. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 897–955, 1988.

[Frank & Vernon 93] M. I. Frank and M. K. Vernon. A hybrid shared memory / message passing parallel machine. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I:232–237, August 1993.

[Fu et al. 92] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *25th Annual International Symposium on Microarchitecture*, pages 102–110, 1992.

[Gharachorloo et al. 90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[Gupta 89] R. Gupta. The fuzzy barrier: A mechanism for the high speed synchronization of processors. In *Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, 1989.

[Hagersten 92a] E. Hagersten. DDM — a cache-only memory architecture. *Computer*, pages 44–54, September 1992.

[Hagersten 92b] E. Hagersten. *Toward Scalable Cache Only Memory Architectures*. PhD dissertation, Swedish Institute of Computer Science, October 1992. SICS Dissertation Series 08.

[Hatcher & Quinn 91] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[Hatcher et al. 91] P. J. Hatcher, M. J. Quinn, and B. K. Seevers. Implementing a data-parallel language on a tightly coupled multiprocessor. In *Proc. 3rd Workshop Programming Languages Compilers Parallel Computers*, 1991.

[Henry & Joerg 92a] D. S. Henry and C. F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, October 1992.

[Henry & Joerg 92b] D. S. Henry and C. F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, October 1992.

[HPFF 93] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, May 1993.

[Hutto & Ahamad 90] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of 10th International Conference on Distributed Computing Systems*, pages 302–309, 1990.

[Intel 91a] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[Intel 91b] Intel Supercomputer Systems Division. *A Touchstone DELTA System Description*, February 1991.

[Jordan 87] H. Jordan. The Force. Technical Report ECE 87-1-1, Dept. of Electrical and Computer Engineering University of Colorado, January 1987.

[Jouppi 93] N. P. Jouppi. Cache write policies and performance. In *Proceedings of 20th International Symposium on Computer Architecture*, pages 191–201, 1993.

[Ju & Dietz 91] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fourth Workshop on Languages and Compilers for Parallelism*. Springer Verlag, August 1991.

[Klaiber & Frankel 93] A. C. Klaiber and J. L. Frankel. Comparing data-parallel and message-passing paragigms. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II:11–II:20, 1993.

[Klaiber & Levy 91] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of 18th International Symposium on Computer Architecture*, pages 43–53, May 1991.

[Koelbel & Mehrotra 91] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[Konstantinidou & Snyder 91] S. Konstantinidou and L. Snyder. Chaos router: Architecture and performance. In *Proceedings of 18th International Symposium on Computer Architecture*, pages 212–221, May 1991.

[Kranz et al. 93] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of Fourth SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, 1993.

[KSR 92] Kendall Square Research. *KSR-1 Technical Summary*, 1992.

[Kuskin et al. 94] 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. To appear in *Proceedings of 1994 International Symposium on Computer Architecture*, 1994.

[Lenoski et al. 92] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[Leung & Zahorjan 93] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, July 1993.

[Lin & Snyder 90] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II:163–170, August 1990.

[MacDonald & Barrusio 94] T. MacDonald and R. Barrusio, 1994. Personal Communication.

[Martonosi & Gupta 89] M. Martonosi and A. Gupta. Tradeoffs in message-passing and shared-memory implementations of a standard cell router. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages III:88–96, August 1989.

[McCreight 84] E. McCreight. The dragon computer system: An early overview. Technical report, Xerox Corp., September 1984.

[Mellor-Crummey & Scott 91] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, pages 21–65, February 1991.

[Minzer 89] S. E. Minzer. Broadband ISDN and Asynchronous Transfer Mode (ATM). *IEEE Communications Magazine*, 27(9):17–24,57, September 1989.

[Mowry & Gupta 91] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, pages 87–106, June 1991.

[Ngai & Seitz 89] J. Y. Ngai and C. L. Seitz. A framework for adaptive routing in multicomputer networks. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 1–9, 1989.

[Ngo & Snyder 92] T. Ngo and L. Snyder. On the influence of programming models on shared memory computer performance. In *Scalable High Performance Computing Conference*, pages 284–291, 1992.

[Nikhil 90] R. S. Nikhil. Id version 90.0 reference manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, September 1990.

[Pierce 88] P. Pierce. The NX/2 operating system. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[Quinn et al. 88] M. J. Quinn, P. J. Hatcher, and K. C. Jourdenais. Compiling C* programs for a hypercube multicomputer. In *Proc ACM/SIGPLAN PPEALS*, pages 57–65, 1988.

[Reinhardt et al. 94] S. Reinhardt, J. Larus, and D. Wood. Typhoon: A user-level shared-memory system. To appear in *Proceedings of 1994 International Symposium on Computer Architecture*, 1994.

[Rose & Steele Jr. 87] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing*, volume ii, pages 2–16, 1987.

[Rosing et al. 90] M. Rosing, R. Schnabel, and R. Weaver. The Dino parallel programming language. Technical Report CU-CS-457-90, Dept. of Computer Science, University of Colorado, April 1990.

[Sequent 87] Sequent Computer Systems Incorporated. *Symmetry Technical Summary*, rev. 1.5 edition, 1987.

[Snyder 92] L. Snyder. Chaos router: finally, a practical adaptive router? In *Parallel Architectures and Their Efficient Use. First Heinz Nixdorf Symposium Proceedings*, pages 146–155, November 1992.

[Stenstrom et al. 93] P. Stenstrom, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of 20th International Symposium on Computer Architecture*, pages 109–118, 1993.

[Su et al. 93] E. Su, D. J. Palermo, and P. Banerjee. Automating parallelization of regular computations for distributed-memory multicomputers in the Paradigm compiler. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II:39–46, 1993.

[Thapar et al. 93] M. Thapar, B. Delagi, and M. J. Flynn. Linked list cache coherence for scalabe shared memory. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 34–43, 1993.

[Thekkath et al. 93] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Efficient support for multicomputing on ATM networks. Technical Report 93-04-03, University of Washington, Department of Computer Science & Engineering, Seattle, WA 98195, 1993.

[TMC 90] Thinking Machines Corp., 245 First St., Cambridge MA 02142. *C* Programming Guide, Version 6.0*, November 1990.

[TMC 91a] J. Frankel. C* language reference manual. Technical report, Thinking Machines Corp., 245 First St., Cambridge MA 02142, 1991.

[TMC 91b] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.

[TMC 92] Thinking Machines Corp. *CMMD 2.0 Reference Manual*, 1992.

[von Eicken et al. 92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[von Hanxleden et al. 92] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings*, pages 97–111, August 1992.

[Wu et al. 91] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.