Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines

Arvind Krishnamurthy*, Klaus E. Schauser†, Chris J. Scheiman†,
Randolph Y. Wang*, David E. Culler*, and Katherine Yelick*

Abstract

Large-scale parallel machines are incorporating increasingly sophisticated architectural support for user-level messaging and global memory access. We provide a systematic evaluation of a broad spectrum of current design alternatives based on our implementations of a global address language on the Thinking Machines CM-5, Intel Paragon, Meiko CS-2, Cray T3D, and Berkeley NOW. This evaluation includes a range of compilation strategies that make varying use of the network processor; each is optimized for the target architecture and the particular strategy. We analyze a family of interacting issues that determine the performance tradeoffs in each implementation, quantify the resulting latency, overhead, and bandwidth of the global access operations, and demonstrate the effects on application performance.

1 Introduction

In recent years several architectures have demonstrated practical scalability beyond a thousand microprocessors, including the nCUBE/2, Thinking Machines CM-5, Intel Paragon, Meiko CS-2, and Cray T3D. More recently, researchers have also demonstrated high performance communication in Network of Workstations (NOW) using scalable switched local area network technology [28, 6, 12]. While the dominant programming model at this scale is message passing, the primitives used are inherently expensive, due to buffering and scheduling overheads [29]. Consequently, these machines provide varying levels of architectural support for communication in a global address space via various forms of memory read and write.

We developed the Split-C language to allow experimentation with new communication hardware mechanisms by involving the compiler in the support for the global address operations. Global memory operations are statically typed, so the Split-C compiler can generate a short sequence of code for each potentially remote operation as required by the specific target architecture. We have developed multiple highly optimized versions of this compiler, employing a range of code-generation strategies for machines with dedicated "network processors." In this study, we use this spectrum of run-time techniques to evaluate the performance trade-offs in architectural support for communication found in several of the current large-scale parallel machines.

*Computer Science Division, University of California, Berkeley, CA 97420, {arvindk,culler,rywang,yelick}@cs.berkeley.edu

†Department of Computer Science, University of California, Santa Barbara, CA 93106, {schauser,chriss}@cs.ucsb.edu

We consider five important large-scale parallel platforms that have varying degrees of architectural support for communication: the Thinking Machines CM-5, Intel Paragon, Meiko CS-2, Cray T3D, and Berkeley NOW. The CM-5 provides direct user-level access to the network, the Paragon provides a network processor (NP) that is symmetric with the compute processor (CP), the Meiko and NOW provide an asymmetric network processor that includes the network interface (NI), and the T3D provides dedicated hardware, which acts as a specialized NP for remote reads and writes. Against these hardware alternatives, we consider a variety of implementation techniques for global memory operations, ranging from general purpose active message handlers to specialized handlers executing directly on the NP or in hardware. This implementation exercise reveals several crucial issues, including protection, address translation, synchronization, responsiveness, and flow-control, which must be addressed differently under the different regimes and contribute significantly to the effective communication costs in a working system.

Our investigation is largely orthogonal to the many architectural studies of distributed shared memory machines, which seek to avoid unnecessary communication by exploiting address translation hardware to allow consistent replication of blocks throughout the system [16, 17, 18, 20], and operating system studies, which seek the same end by extending virtual memory support [21, 1, 8]. In these efforts, communication is caused by a single load or store instruction, and the underlying hardware or operating system mechanisms move the data transparently. We focus on what happens when the communication is necessary. So far, distributed shared memory techniques have scaled up from the tens of processors toward a hundred, but many leaders of the field suggest that the thousand processor scale will be reached only by clusters of these machines in the foreseeable future. Our investigation overlaps somewhat with the cooperative shared memory work, which initiates communication transparently, but allows remote memory operations to be serviced by programmable handlers on dedicated network processors [18, 24]. The study could, in principle, be performed with other compiler-assisted shared memory implementations [31], but these do not have the necessary base

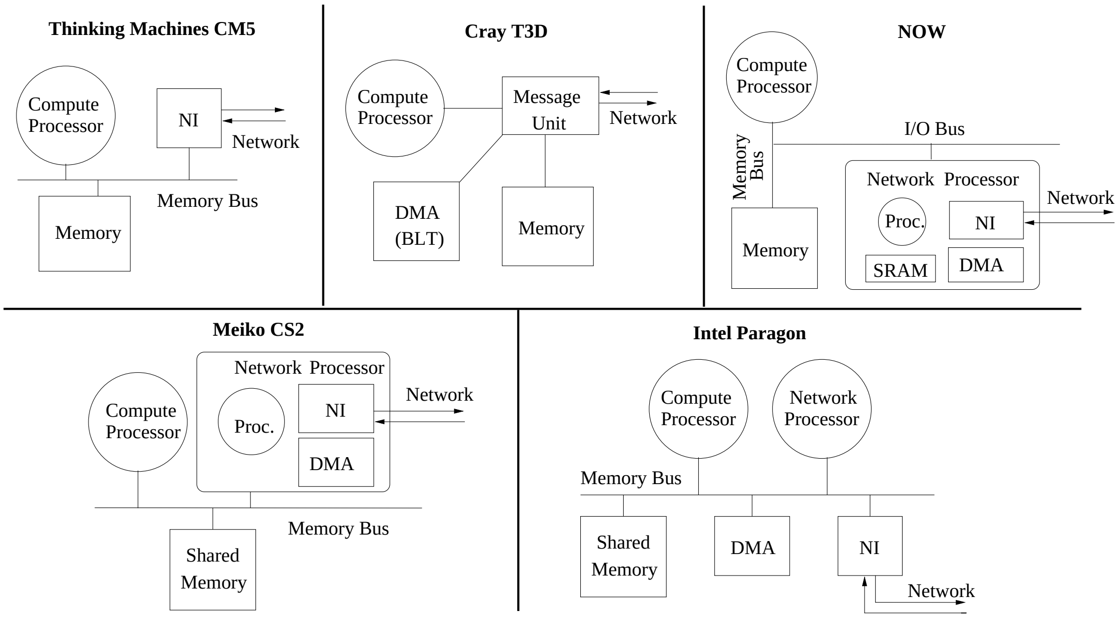

Figure 1: *Structure of the multiprocessor nodes.*

of highly optimized implementations on a range of hardware alternatives.

The rest of the paper is organized as follows. Section 2 provides background information for our study. We briefly survey the target architectures and the source language. Section 3 sketches the basic implementation strategies. Section 4 provides a qualitative analysis of the issues that arise in our implementations and their performance impacts. Section 5 substantiates this with a quantitative comparison using microbenchmarks and parallel applications. Finally, Section 6 draws conclusions.

2 Background

In this section we establish the background for our study, including the key architectural features of our candidate machines and an overview of Split-C.

2.1 Machines

We consider five machines, all constructed with commercial microprocessors and a scalable, low-latency interconnection network. The processor and network performance differs across the machines, but more importantly they differ in the processor's interface to the network. They range from a minimal network interface on the CM-5 to a full-fledged processor on the Paragon. Figure 1 gives a sketch of the node architecture on each machine.

Thinking Machines CM-5: The CM-5 [19] has the most primitive messaging hardware of the five machines. Each node contains a single 33 MHz Sparc processor and a conventional MBus-based memory system. (We ignore the vector units in both the CM-5 and Meiko machines.) The network interface unit provides user-level access to the network. Each message has a tag identifying it as a system message, interrupting user message, or non-interrupting user message that can be polled from the NI. The compute processor sends messages by writing to output FIFOs in the NI using uncached stores; it polls for messages by checking network status registers. Thus, the network is effectively a distributed set of queues. The queues are quite shallow, holding only three 5-word messages. The network is a 4-ary fat tree that has a link bandwidth of 20 MB/sec in each direction.

Intel Paragon: In the Paragon [14], each node contains one or more compute processors (50 MHz i860 processors) and an identical CPU dedicated for use as a network processor. Our configuration has a single compute processor per node. The compute and network processors share memory over a cache-coherent memory bus. The network processor, which runs in system mode, provides communication through shared memory to user level on the compute processor. It is also responsible for constructing and interpreting message tags. Also attached to the memory bus are 2 DMA engines and a network interface. The network interface provides a pair of relatively deep input and output FIFOs (2KB each), which can be driven by either processor or by the DMA engines. The network is a 2D mesh with links operating at 175 MB/s in each direction.

Meiko CS-2: The Meiko CS-2 [4] node contains a special-purpose "Elan" network processor integrated with the network interface and DMA controller. The network processor is attached to the memory bus and is cache-coherent with the compute processor, which is a 40 MHz three-way superscalar SuperSparc processor. The network processor functions both as a processor and as a memory device, so the compute processor can issue commands to the network interface and get status back via a memory exchange instruction at user level. The network processor has a dedicated connection to the network; however, it has only modest processing power and no general purpose cache, so instructions and data are accessed from main memory. The network is comprised of two 4-ary fat-trees that have a link-level bandwidth of 70 MB/sec.

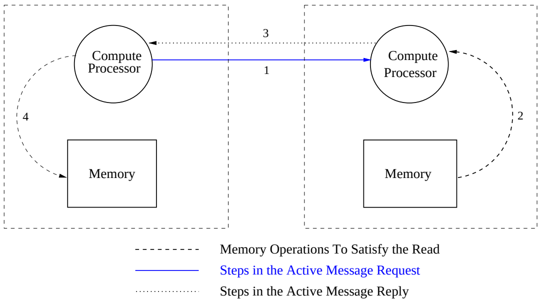

Figure 2: *Handlers are executed on the CP.*

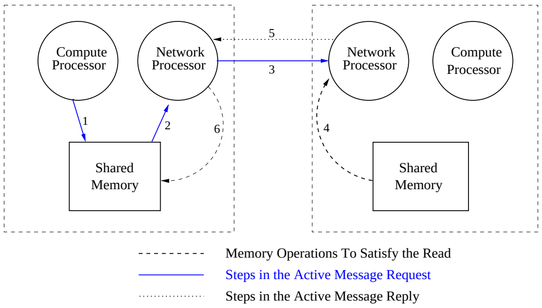

Figure 4: *Handlers are executed on the NP.*

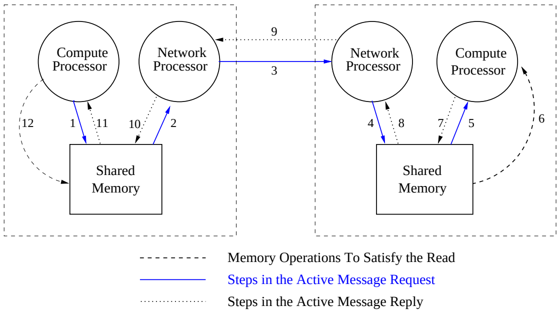

Figure 3: *NPs are treated simply as network interfaces.*

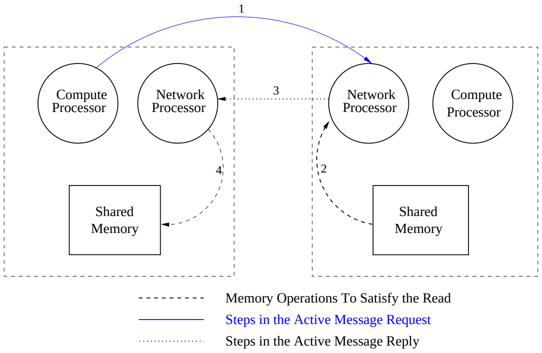

Figure 5: *CP directly injects messages into the network.*

Cray T3D: The Cray T3D [10] has a sophisticated message unit, which is integrated into the memory controller, to provide direct hardware support for remote memory operations. A node consists of a 150 MHz Alpha 21064 [27] processor, memory, and a "shell" of support circuitry to provide global memory access and synchronization. A remote memory operation typically requires a short sequence of instructions, to set up the destination processor number in an external address register, issue a memory operation, and then test for completion, rather than a simple load or store. The shell also provides a system-level bulk transfer engine, which can DMA large blocks of contiguous or strided data to or from remote memories. Processors are grouped in pairs, share a network interface and block-transfer engine, and all 2-processor nodes are connected via a three-dimensional torus network with 300 MB/s links.

Berkeley NOW: The Berkeley-NOW [2] is a cluster of UltraSparc workstations connected together by Myrinet [6]. The CP is a 167 MHz four-way super-scalar UltraSparc processor. The Myrinet NI is an I/O card that plugs into the standard SBus. It contains a 32-bit CISC-based "LANai" network processor, DMA engines, and local memory (SRAM). The CP can access the NP's local memory through uncached loads/stores on memory mapped addresses. The NP can access the main memory through special DMA operations initiated through the I/O bus. The Myrinet network is composed of crossbar switches with eight bidirectional ports, and the switches can be linked to obtain arbitrary topologies. The network provides a link bandwidth of 80 MB/sec.

2.2 Global address space language

Many parallel languages, including HPF [15], Split-C [13], CC++ [9], Cid [23], and Olden [7], provide a global address space abstraction built from a combination of compiler and runtime support. The language implementations differ in the amount of information available at compile time and the amount of runtime support for moving and caching values. We consider the problem of implementing a minimalist language, Split-C [13], which focuses attention on the problems of naming, retrieving, and updating remote values.

A program is comprised of a thread of control on each processor from a common code image (SPMD). The threads execute asynchronously, but may synchronize through global accesses or barriers. Processors interact through reads and writes on shared data. The type system distinguishes local accesses from global accesses, although a global access may be to an address on the local processor.

Split-phase (or non-blocking) variants of read and write, called *get* and *put*, are provided to allow the long latency of a remote access to be masked. These operations are completed by an explicit *sync* operation. Another form of write, called *store*, avoids acknowledging the completion of a remote write, but rather increments a counter on the processor containing the target address. This operation supports efficient one-way communication and remote event notification. Bulk transfer within the global address space is provided in both blocking and non-blocking forms. In addition to read and write, *atomic* read-modify-write operations, such as fetch-op-store, are supported.

3 Implementations

Global memory operations fundamentally involve three processing steps: (1) the processor issues a request, (2) the request is served on a remote node by accessing memory, possibly updating state or triggering an event, and (3) a response or completion indication is delivered to the requesting node. Between these steps, information flows from the CP through network processors (NPs), network interfaces (NIs), and memory systems. Consequently, a basic issue in any implementation is choosing which set of hardware elements are involved in the transfer and where the message is handled. In this section, we outline four possible strategies for implementing global memory operations.

Compute Processor as Message Handler (Proc): In our first approach, the message handlers are executed on the CP. In the simplest implementation of this strategy, the CP injects the message into the network (through the network interface), and the CP on the remote node receives the message, executes the handler, and responds (as shown in Figure 2 for a remote read operation).

The same basic strategy can be employed with network processors, using them simply as smart network interfaces (as shown in Figure 3). The communication between a CP and an NP on a node occurs through a queue built in shared memory. The CP writes to the queue, and the NP, which is constantly polling the queue, retrieves the message from memory and injects it into the network. The message is received by the NP on the remote side, enqueued into shared memory, and eventually handled by the remote CP, whereupon a handler executes and initiates a response. In this implementation, the network processor's task is to move data between shared memory and the network, guarantee that the network is constantly drained of messages, and perform flow control by tracking the number of outstanding messages.

Network Processor as Message Handler (NP): In our next approach, the message handlers are executed on the NPs thereby reducing the involvement of the CPs (see Figure 4). The request initiation is similar to the base implementation; the CP uses shared memory to communicate the message to the NP, which injects the message into the network. The NP on the remote node receives the message, executes the corresponding handler, and initiates a response without involving the CP. The NP on the requesting node receives the response and updates state to indicate the completion of the remote request without involving the CP. This strategy streamlines message handling by eliminating the involvement of the compute processors.

Message Injection by the Compute Processor (Inject): In this approach, the CP on the requesting node directly injects messages into the network without involving the NP (as shown in Figure 5). The remote NP receives and executes the message before initiating the response, which is eventually received and handled by the NP on the requesting node. This approach streamlines message injection by eliminating the NP's involvement. However, in this strategy, since both the CP and the NP can inject messages into the network, the network interface must be protected to ensure mutually exclusive access.

Message Receipt by either Processor (Receive): Our next approach differs from the Inject strategy in one aspect; the compute processors are also allowed to receive and handle the messages. As with the Inject strategy, the CP directly injects requests into the network. However, on the remote node, the request is serviced by either the CP or the NP depending on which processor is available. Similarly, when the reply comes back, it is handled by either one of the two processors on the source node. In the Inject approach, during a global memory operation, there are three points at which an NP is involved in a message send/receive while the CP is involved only once. Since interfacing with the NI is typically expensive, this asymmetry in roles could potentially lead to a load imbalance during a global communication phase. The Receive strategy corrects this asymmetry by dynamically balancing the load between the CP and the NP.

We have implemented ten versions of the Split-C language. Each version is highly optimized for the underlying hardware. On the CM-5, which has no network processors, a sole version exists. All message handlers are executed directly by the compute processors. On the Meiko and the NOW, we have an implementation that executes the message handlers on the CP and another that executes them on the NP [25, 26]. The CP does not interact with the network directly in either case. On the Paragon, we have an implementation for each of the four different message handler placement strategies discussed in this section. On the T3D, we have implemented a sole version that handles all remote memory accesses using the combined memory controller and network interface [3].

4 Issues and Qualitative Analysis

In this section we present a set of interacting issues that shape the implementation of global address operations. Our goal is to identify the hardware and software features that affect both correctness and performance. This provides a framework for understanding the performance measurements in the next section.

There are three dimensions to communication performance. The *latency* of an operation is the total time taken to complete the operation. If the program must wait until the operation completes, this is the most important figure of merit. However, if the program has other work to do, then the important measure is *overhead*, which is the processing time spent issuing and completing a communication event. If there are many concurrent communication events occurring at the same time, then the time for each event to get through the bottleneck in the communication system, called the occupancy or *gap*, may matter most, since it determines the effective communication bandwidth. Our perspective is influenced by the LogP model [11], although we use a different definition of latency, which includes overhead.

We divide the set of implementation issues into three categories: those that are determined primarily by the machine architecture, those that are determined by the particular language implementation, and those that are only determined by application usage characteristics.

4.1 Architectural issues

We begin with the family of architectural issues in a bottom-up fashion, starting with the basic factors involved in moving bits around. We then consider the processors themselves, and work upwards toward system issues of protection and address translation.

4.1.1 Interface to the network processor and the network interface

The interface between a processor and the network is a notorious source of software and hardware costs. We consider two design goals in this section, minimizing overhead and minimizing latency, which suggest opposing designs.

To minimize latency, we should streamline the communication process by reducing the number of memory transfers between processors. One expects the latency to be minimized if requests are issued directly from the CP into the network, the remote operation is handled and serviced directly out of the network, and the response is given directly to the requesting processor.

To minimize the communication overhead for the CP, the solution is to offload as much work as possible to a separate NP. The offloaded work of transferring data into (or out of) the network involves checking various status indicators, writing (or reading) FIFOs, and checking that the transfer was successful. On all five machines this transfer is expensive. Thus, we would expect the reduction in overhead to be significant as the cost of transfer to the network is traded off against communication with the NP.

Surprisingly, on the Paragon, the cache coherency protocol (between the CP and the NP) results in at least four bus transfers for an eight-word processor-to-NP transfer, at a cost greater than a processor-to-NI transfer [22]. On the Meiko, because the NI is integrated into the NP, the CP can not directly send into the network. Communication between the processor and NP involves a transfer request being "written" to the NP with a single exchange instruction, whereupon the NP pulls the message data directly out of the processor's cache and writes it into the network. The NOW configuration is similar to that of Meiko's with the NP having a dedicated connection to the NI. The difference is that the CP and the NP do not share memory. Instead, the CP and the NP communicate through a queue located in the NP's local memory, which is accessible to the CP through memory mapped addresses. On the T3D, crossing the processor chip boundary is the major cost of all remote memory operations [3]; however, the external shell is designed specifically to present the network as conveniently as possible to the processor.

A correctness issue also arises if the processor injects requests directly into the network and the NP is to handle requests and inject responses; the network interface must be protected to ensure mutually exclusive access. This issue does not arise on the Meiko or NOW, since they do not allow direct access to the NI from the processor. In our Paragon implementation, explicit locks are used to guarantee exclusive access.

4.1.2 Relative performance and capability

One of the stark differences in our target platforms is the processing power and capabilities of the NPs. One expects performance to improve if handlers are executed on the NP. However, if the NP is much slower than the processor, as on the Meiko, then it may be best for the NP to do only simple operations; it may be faster to have the NP pass complex requests to the processor than to execute the request itself. The optimal choice depends not only on the relative speed of the two processors, but also their relative load; if the processor is busy, then the best performance may be achieved by executing the handler on the slower NP. Of course, the request must always be passed on to the remote processor if it demands capabilities not present in the NP. For example, the Meiko NP does not directly support floating-point or byte operations, the NOW NP has no floating point support, and the T3D shell can only serve remote memory accesses and very limited synchronization operations.

4.1.3 Protection

Protection issues arise at each step of a global access operation. The network interface must be protected from incorrect use by applications. Messages sent by one application must arrive only at target remote processes for which it is authorized. Hosts must continue to extract messages to avoid network deadlock. Finally, handlers must only access storage resources that are part of the application. The traditional solution in LANs and early message passing machines was to involve the operating system on sending and receiving every message (Ncube/2 [29]). This requirement can be eliminated with more complete architectural support, described below, and with coarser system measures, such as gang scheduling of applications on partitions of the machine.

The Paragon has primitive hardware support for protection. It does not distinguish user and systems messages or messages from different processes, and there is no safe user-level access to the NI. Instead of invoking the OS for each communication operation, protection is enforced by passing all messages through a shared buffer to the NP, which runs at system level and provides protection checks, resource arbitration, and continually drains the network. In all but the base implementation, remote operations are serviced directly on the NP, which performs protection checks on access to user addresses. In the implementations in which the CP directly injects requests into the NI, there is a protection loophole; these experiments reflect the performance that would be possible if the Paragon were to adopt measures similar to the CM-5 or Meiko.

On the CM-5, the NI attaches a tag to each message so that user and system messages can be distinguished. The NI state is divided into distinct user and system regions, so that the parallel application can only inject messages for other processes within the application. Gang-scheduling on a "subtree" of the network is used to ensure that applications do not interfere with each other. Also, since remote services are only performed on the processor, the normal address translation mechanism enforces protection. The NOW also gang-schedules parallel jobs. It does not support time-slicing currently, so a parallel program's traffic is insulated from others. The control program on the NP is protected from user tampering, and it performs protection checks on user supplied addresses.

The Meiko protection mechanism is more sophisticated. A collection of communicating processes possess a common communication capability. All messages are tagged with the capability, which is used to identify the communication context on the NP. This context includes the set of remote nodes to which the application is authorized to send messages and the virtual memory segment of the local process that is accessible to remote processes. Time-slicing the NP processor allows applications to make forward progress, so arbitrary user handlers may be run directly on the NP.

The T3D enforces protection entirely through the address translation mechanism. If an access is made to an authorized remote location, the virtual to physical translation will succeed and the shell will issue a request access to the designated physical location on the appropriate physical node. Parallel applications are gang-scheduled on a "subcube" of the machine so they don't compete for network resources.

4.1.4 Address translation

A key issue in all of our implementations is how the address of a globally accessible location is translated. A global pointer is statically distinct from a local pointer, so the address translation is potentially a joint effort by the compiler-generated code, the remote handler, and the hardware. If remote operations are served by the CP, the virtual to physical translation is performed by the standard virtual memory mechanism, and page-fault handling is decoupled from communication. If remote operations are served by the NP, many alternatives arise.

On the Meiko the user thread on the NP runs in the virtual address space of the user process. (The NP contains its own TLB, and its page table is kept consistent with that used by the processor.) Thus, the address translation on request issue is the same as for the Proc strategy.

On the Paragon, if the message handler is run on the NP, it runs in kernel mode. In the current version of OSF/AD, it executes directly in the physical address space. Thus, all accesses to user address space are translated and checked in software on the NP. In principle a TLB could be used to accelerate this translation, but since the nodes are potentially time-sliced, the NP would still need to check that it contained a mapping for the target process of the message and adjust its mapping or emulate the context as appropriate. The remote page-fault issue arises as well, but the message is explicitly aborted by the handler.

On the NOW, since the NP is attached to the I/O bus, it can access main memory only through valid I/O space addresses. However, since the I/O bus supports only 28-bit addresses, only a portion of the user's address space can be mapped into the I/O space at any given time. Consequently, if an access is made to an address that is not currently part of the I/O space, the NP passes the request to the compute processor.

The T3D takes a completely different approach in that the virtual address is translated on the processor that issues the request. The page tables are set up so the result of the address translation contains the index of a remote processor in a small, external set of registers and a physical address on that node. It is possible that a valid address for the remote node causes an address fault on the node issuing the request, if the remote node has extended its address space beyond that of the requester. The language implementation avoids this problem by coordinating memory allocation. No paging is supported.

4.1.5 DMA support

All of the issues above come together in DMA support for bulk transfer operations. On the Paragon and Meiko, DMA offers much greater transfer bandwidth than small messages. The Paragon operates on physical addresses and has a complex set of restrictions for correct operation, so it must be managed at kernel level. To improve overall network utilization, the NP fragments large transfers into page sized chunks. The Meiko provides more sophisticated DMA support by allowing the user to specify arbitrary sized transfers on virtual addresses. The DMA engine is part of the NP and automatically performs address translation and fragments the transfer into 512 byte chunks, which are interleaved with other traffic. The T3D "block transfer engine" operates on physical addresses and is only accessible at kernel level, so the cost of a trap is paid on startup. On the NOW, a bulk transfer requires the participation of two DMA engines; the host DMA moves data between main memory and the NP's local memory, while the NI DMA is responsible for moving data between the NP's local memory and the network. It operates in the I/O address space and requires an additional copy to (or from) the user space.

4.2 Language implementation issues

We now turn to the family of issues at the language implementation level, given that the network processor can execute handlers and has specific architectural characteristics that can be fully exploited.

4.2.1 Generality of handlers

If the network processor runs in kernel mode, as on the Paragon, it can only run a fixed set of "safe" handlers.[1] The protection and address translation capabilities of the Meiko NP make general handlers possible, but its poor performance makes it usable only for highly specialized handlers. Atomic handlers are challenging to implement efficiently on the NP. Expensive locking may be required to ensure exclusive access to program states by the CP and NP. Our approach is to always execute the complex atomic operations on the compute processor to avoid costly locking.

4.2.2 Synchronization

Non-blocking operations, such as get, put, and store require some form of synchronization event to signal completion. This is easily implemented with counters, but if operations are issued by the processor and handled by the NP, then the counters must be maintained properly with minimal cost for exclusive access. An efficient solution is to split the counter into two counters, using one counter for the compute processor increments and the other for the network processor decrements. No race condition can occur since each processor can only write to one location, and can read both. The sum of the two counters produces the desired counter value. Implementing this on the Meiko and the Paragon is best accomplished by having each half of the counter in a separate cache line to avoid false sharing.

4.2.3 Optimizing for specialized handlers

If some of the handlers are to be executed on the NP, they can be optimized for their task and the specific capabilities of the NP. On the Paragon, the translations of frequently accessed variables, e.g., the completion counters, can be cached for future use. Message formats are specialized for the NP handlers to minimize packet size. One-way operations, such as stores, can be treated specially to reduce the number of reverse acknowledgments needed for flow control. The Meiko allows for optimizations of a different sort, as the handler code can be mapped directly onto some specialized operations supported by the NP, such as remote atomic writes [26].

4.3 Application issues

Considering architectural and language implementation issues in isolation, one can construct a solution that attempts to minimize the latency, overhead, and gap for the individual global access operations, striking some balance between the three metrics. However, the effective performance of these operations depends on how they are actually used in programs. Two issues that have emerged clearly in this study are responsiveness and the frequency of remote events.

[1] This set might be enlarged by using sandboxing or software fault isolation techniques [5, 30].

Feature	Operation	CM-5 (Proc)	Meiko Proc	Meiko NP	Paragon Proc	Paragon NP	Paragon Inject	Paragon Receive	T3D (NP)	NOW Proc	NOW NP
RT Latency	AM	14.0	23	52	20.1	20.1	15.3	12.0	9.6	28.9	17.7
	Read	16.5	24.3	32.9	20.5	15.9	13.2	12.2	0.85	29.5	21.6
	Write	14.6	24.3	13.8	20.4	15.3	12.1	12.1	0.98	29.3	19.4
Overhead	Get	6.0	2.6	7.1	5.9	2.4	2.7	3.1	0.40	7.1	2.0
	Put	6.1	2.5	4.1	5.0	2.3	2.6	3.1	0.31	7.1	2.1
	Store	3.4	1.7	5.4	5.2	2.4	2.7	3.1	2.9	2.6	2.0
Gap	Get	6.2	15.5	36.1	8.3	8.4	6.7	8.0	0.40	7.9	9.2
	Put	6.0	15.8	21.3	8.5	7.5	6.7	7.3	0.31	7.9	8.8
	Store	3.3	13.9	20.8	8.5	7.3	2.9	4.8	2.9	5.8	8.6
Gap (to 2)	Get	6.1	15.7	24.7	9.0	8.4	6.7	7.4	0.40	15.7	18.4
	Put	5.9	15.6	20.1	8.5	7.6	5.7	7.3	0.31	15.7	17.7
	Store	3.4	14.0	19.8	8.2	7.5	2.9	3.2	2.9	11.5	11.7
Gap (exchange)	Get	9.3	28.4	47.2	16.0	13.5	13.2	11.9	0.56	15.7	15.2
	Put	9.3	27.8	23.8	15.8	12.7	12.4	11.3	0.37	15.7	13.9
	Store	7.3	15.4	24.2	15.6	14.1	8.0	7.5	4.7	8.9	10.3

Table 1: *Basic Split-C operations, for different versions of Split-C (times in us). Proc indicates the compute processor implementation, NP the network processor implementation, Inject the implementation where the compute processor directly injects messages, and Receive where it also directly receives responses.*

4.3.1 Responsiveness

The prompt handling of incoming messages is important for minimizing latency. One way to ensure messages are handled as they arrive is for the message to trigger an interrupt. Unfortunately, few commodity processors have fast interrupts, so where possible we utilize polling of the network interface. If the CP is responsible for polling and fails to do so because it is busy in compute intensive operations, the effective latency can increase dramatically. If remote operations are handled on the NP, it can be responsive to these requests, regardless of the activities of the processor.

4.3.2 Frequency of remote events

Remote events are operations where control information is transmitted to a remote process along with data. The simplest remote event we consider is the signaling store, which informs the remote processor how much data has been stored into it by incrementing a counter. Synchronization operations, such as fetch&add and more general atomic procedures involve more extensive operations within the remote address space. Many event driven applications use a distributed task queue model, where communication causes a new event to be posted on a queue.

If a program invokes very few remote operations, architectural support for communication has very little impact on performance. If a program is communication intensive and if all remote operations are variants of read and write, which involve only the remote memory and do not interact with the remote processor, then, unsurprisingly, devoting hardware to serve these operations will improve performance. However, specific support for simple read and write operations does little to support remote events, such as stores, since they are relatively expensive to implement as multiple operations on the remote memory space. Thus, the latency, overhead, and gap of these remote event operations varies significantly across our platforms, but the effective impact on performance depends on how frequently they are used in applications, which also varies dramatically.

4.4 Summary

Each implementation strategy on each platform must address the issues raised in this section, it represents a particular point of balance in the opposing trade-offs present. In this section, we have examined in qualitative terms how the available architectural support, the language implementation techniques, and the program usage characteristics influence the performance of global access operations. Given this framework, let us examine the performance obtained on each of the operations in isolation and the resulting application performance.

5 Performance Analysis

In this section we present detailed measurements of our ten implementations to provide a quantitative assessment of the issues presented in the previous section. We divide our discussion into four parts. First, we examine the raw communication performance of active messages and Split-C primitives. We also study bulk synchronous communication patterns, where multiple processors simultaneously exchange messages. Next, we examine bulk transfers and the achieved bandwidth. Then we study a microbenchmark that illustrates the impact of attentiveness to the network. Finally, we examine complete applications written in Split-C.

5.1 Performance of Split-C primitives

Table 1 shows the round-trip latency, gap, and overhead for active messages, as well as get, put, and store operations under the various implementations. The upper three groups test a single requester and single remote server. The lower two involve communication among multiple processors.

Round-trip Latency: To evaluate system impact on latency, we consider three types of operations, all of which wait for an acknowledgement of completion. The first is a general active message, measured for the case of a null handler, and the others are blocking memory operations, read and write. The latency measurements exposes two of the issues raised earlier: total round trip time is minimized by avoiding the use of NPs during injection (thereby reducing memory transfers), and the use of specialized hardware to support particular operations significantly improves their performance.

Both the CM-5 and the Paragon show that faster communication is possible when messages are directly injected into the network. We see that the CM-5 latency is small compared to other architectures, since the CP directly injects and retrieves messages from the network. The effect of reducing memory transfers between the CP and the NP is most clearly seen on the Paragon. On the Paragon, each implementation improves on the previous versions. The advantage of Paragon-NP over Paragon-Proc is especially remarkable given the additional overhead of software address translation in Paragon-NP. The further improvement of Paragon-Inject implies that the benefit of directly injecting into the network outweighs the additional cost of locks, which are required for mutually exclusive access to the NI. On the NOW, the NP strategy eliminates the message transfer between the NP and the CP; however, the NP can access the CP's memory only through expensive DMA operations issued over the I/O bus. The lower latencies for the NP implementation imply that eliminating the message transfer more than compensates for the cost of the DMA operation.

The Meiko-NP implementation has higher latencies for active messages and reads, since the NP is much slower than the CP. However, the write operation on Meiko-NP is much faster than the read, since the Meiko has hardware support for remote write. Similarly, on the T3D, reads and writes are much faster than active messages, since reads and writes are directly supported in hardware, while active message operations must be constructed from a sequence of remote memory operations [3]. In contrast, the four Proc implementations of read and write (on the CM-5, Meiko, Paragon, and NOW) are built using active messages, so they take the time of a null active message plus the additional time needed to read or write.

The round-trip measurements on the Paragon also bring out protection and address translation issues. The Paragon Proc and Paragon NP implementations show no difference in latency for active messages, because user supplied handlers cannot be executed on the NP due to inadequate protection. For the Paragon-NP and Inject versions, reads are slower than writes because a read reply requires an extra address translation step for storing the value read into a local variable. (In the Receive implementation, the difference between the read and write costs disappears since the replies are handled on the CP.)

Overhead: One expects that the overhead when writing directly to the NI will be greater than if an NP is involved; surprisingly, this is not always the case. Contrary to expectation, the overhead costs for the NP and the Inject versions on the Paragon are similar, which implies that it is as efficient to write to the NI as writing to shared memory on this platform. As expected, the Paragon and the NOW implementations that use the NP for message handling have much less overhead, since the CP does not handle the reply. On the other hand, the Meiko-NP overhead is higher than the Meiko-Proc, because of an implementation detail. To avoid constant polling on the NP, the more expensive event mechanism is used instead of shared memory flags for handing off the message to the NP. On the CM-5, we observe that the sending and receiving overhead accounts for almost all of the round-trip latency, unlike the other implementations that involve a co-processor.

Comparing the store overhead to the other results helps reveal the underlying architecture. As expected, the CM-5 overhead for gets and puts is almost twice the overhead for stores, which do not require a reply. On the T3D, the overhead for gets and puts is almost half the latency for read and writes, which means that pipelining two or three operations is sufficient to hide the latency. Unlike other platforms, stores on the T3D have a much higher overhead, since the store involves incrementing a remote counter, it cannot be mapped onto T3D's hardware read/write primitives; instead, a general purpose active message is used to implement stores.

Gap: The results for the gap expose the bottlenecks in the various systems. For the CM-5 and the T3D, the gap is the same as the overhead, which indicates that the sending processor is the bottleneck. The higher gap for Meiko-NP and the NOW-NP implementations show us that the slower NP is the bottleneck. On the Paragon, the get operation has a higher gap than puts due to an extra software address translation made while handling the reply. This behavior means that the NP on the sending side is the bottleneck. In the Inject implementation, the cost difference between gets and puts disappears implying that the NP on the remote node has become the bottleneck.

Two other observations can be made concerning the gap results for the Paragon. First, there is no substantial change in the gap between the Proc and NP implementations in spite of removing the compute processor from the critical path. Second, the store gap is lower due to an implementation optimization that bunches together acknowledgments. Note that even though the language does not require acknowledgments for stores, they are sent to ensure flow control in the network (for all versions) and availability of buffer space (for the Proc version).

It is also interesting to note that the NOW-NP implementation trades-off gap for latency by having the NP be responsible for both interfacing with the NI as well as handling the messages. While this approach lowers latency by eliminating messages transfers between the CP and the NP, it increases the load on the NP. If we view the various components of the system as different stages of a pipeline, the NP strategy eliminates some of the stages in the pipeline while increasing the time spent in the longest stage. Consequently, it improves latency at the expense of gap.

Gap-2: We can further isolate the system bottleneck by modifying the gap microbenchmark to issue gets, puts, and stores to two remote nodes. If an operation can be issued more frequently when issued to two different nodes, then the bottleneck for the operation is the processing power of the remote node; otherwise, it is send side limited. For the CM-5 and the T3D, we notice that the operations are send side limited. On the Paragon, the numbers support our earlier conjecture that the NP on the source node is the bottleneck in the NP version while the NP on the destination node is the bottleneck in the Inject and Receive versions. For the Meiko-NP implementation, the remote NP is the bottleneck for gets while the NP on the source node is the bottleneck for puts and stores. Similarly, for the NOW-NP implementation, we observe that the NP on the source node is the bottleneck for gets and puts while the NP on the destination node is the bottleneck for stores.

Gap for Exchange: Our final microbenchmark measures the gap when two processors issue requests to each other simultaneously. This test exposes the issues relating to how

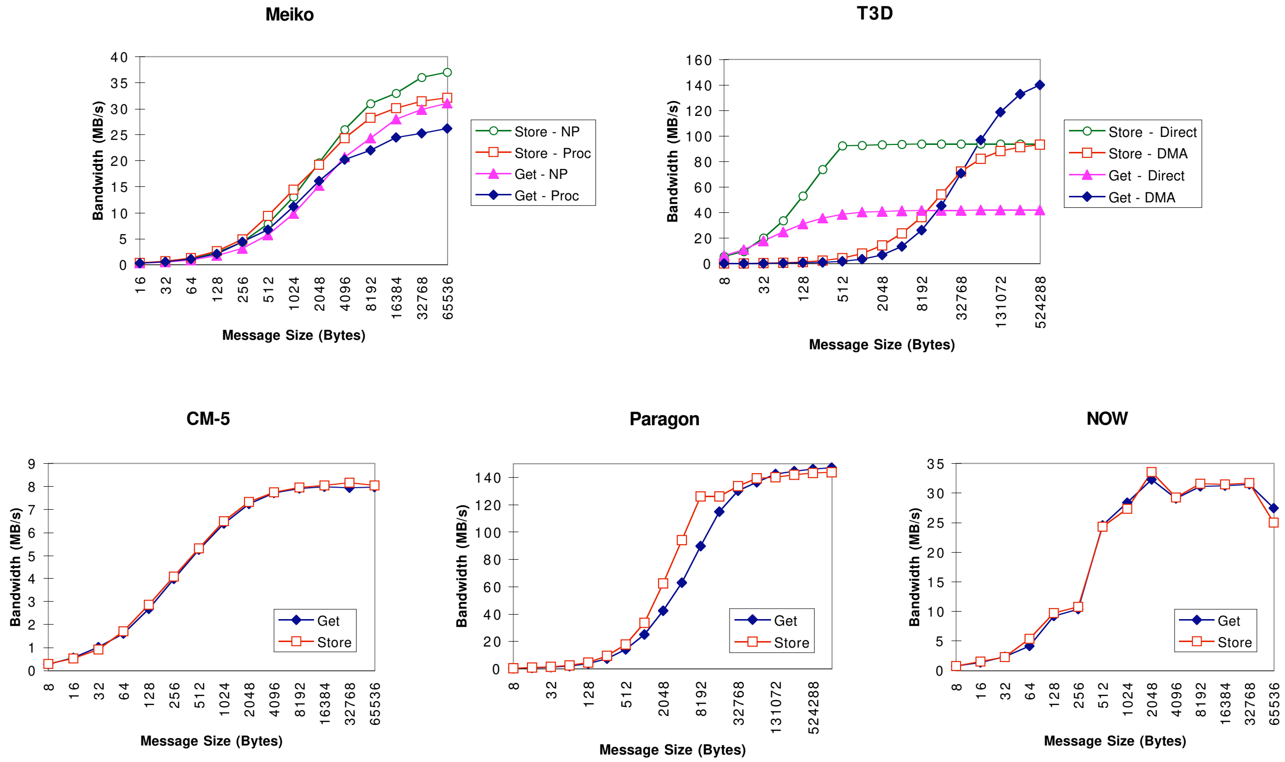

Figure 6: *Bandwidth of Split-C bulk get and store operations for our study platforms.*

the CP and NP divide up the workload involved in communication. As expected, for the Proc implementations on the Paragon and the Meiko, the cost is roughly twice the regular gap since the compute processor experiences the overhead for its message as well as for the remote request. Since the NP and Inject versions allow for the overlapping of resources, the gap costs increase by less than a factor of two. The Receive version, where the compute processor receives and handles messages to share the communication workload with the NP, performs better than the Inject and NP versions in spite of incurring the cost of using locks. It is interesting to observe that the Receive version has higher overhead and gap when a single node issues fetches from a remote node, but it performs better for bulk-synchronous communication patterns, such as exchange.

Conclusion: The latency, overhead, and gap measurements of the Split-C primitives quantify the combined effects of the trade-offs discussed in the previous section. In particular, we can observe the utility of having direct access to the network, a fast NP, and of optimizing the global access primitives onto available architectural support in the target platform. We are also able to observe the effect of factors like software address translation and protection checking.

5.2 Bandwidth for bulk transfers

Figure 6 shows the bandwidth curves for the bulk store and bulk get operations for the different machines. With the exception of the CM-5, all our architectures have a DMA engine to support bulk transfers. The Meiko NP implementation out-performs Meiko Proc for long messages; for stores, it achieves 38 MB/s compared to 32 MB/s. This occurs because the Proc version trades off bandwidth for latency by having the NP continuously poll for new messages from the CP. While this reduces the latency, it constantly takes resources from the NP and reduces the bandwidth for bulk transfers. The NP implementation, on the other hand, only schedules threads on the NP when needed. All the Paragon implementations use the same bulk transfer mechanism (since the device is complex to control and must be operated at kernel level) and achieve the same performance: a maximum bandwidth of 144 MB/s. The CM-5 achieves only 10 MB/s. The T3D provides two different mechanisms for bulk transfer, which differ in startup cost and peak bandwidth, and thus would be employed in different regimes. The NOW throughput is limited by the SBus bandwidth.

5.3 Polling granularity

To study the impact of attentiveness on communication performance, we use a microbenchmark where each processor performs a simple compute/poll loop with the computation granularity varied based on an input parameter; after each computation the process may poll for messages. All processors take turns computing while the remaining processors request a single data item from the busy processor. The requesting processors need this data item to make progress. If this request is not serviced immediately, the requesters idle, and only a single processor is busy computing at any given time. However, if the compute granularity is small or if a NP is used to service requests, the responses come back immediately, and all processors can work in parallel.

Figures 7 and 8 show the impact of varying the compute granularity on the overall run-time. As expected, not polling or polling infrequently results in poor performance. The NP implementation always performs well, because the NP immediately services requests. For a wide range of gran-

Program	Description	Meiko CS-2			Intel Paragon		
		Problem Size	Time (in sec)		Problem Size	Time (in sec)	
			Main proc	NP		Main proc	NP
radix	Radix sort	8 million keys	35.2	55.4	4 million keys	17.3	16.1
sample	Sample sort	16 million keys	20.9	25.3	4 million keys	7.82	6.80
p-ray	Ray-tracer	512x512 tea pot	37.5	38.0	512x512 tea pot	85.4	81.5
sampleb	Sample sort, bulk transfers	16 million keys	7.40	7.43	4 million keys	5.63	5.50
radixb	Radix sort, bulk transfers	16 million keys	15.3	14.4	4 million keys	10.7	10.7
bitonic	Bitonic sort	8 million keys	17.0	16.0	8 million keys	24.2	24.2
fftb	FFT using bulk transfers	8 million pts	13.6	12.5	8 million pts	6.54	6.30
cannon	Cannon matrix multiply	1024x1024 matrix	13.9	12.4	1024x1024 matrix	41.5	41.5
mm	Blocked matrix multiply	128x128 matrix	29.0	23.2	128x128 matrix	47.2	46.1
fft	FFT using small transfers	8 million pts	17.5	13.4	8 million pts	8.12	7.04
shell	Shell sort	16 million keys	44.7	21.7	4 million keys	14.8	14.7
wator	N-body simulation of fish	400 fish	139	34	400 fish	104	93.4

Table 2: *Run times for various Split-C programs on a 16 processor Meiko CS-2 and an 8 processor Paragon. Run times (in seconds) for both the main processor and NP implementations are shown in the table.*

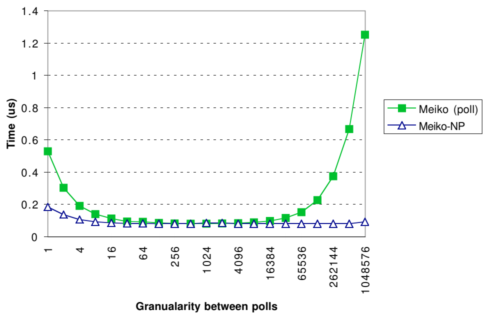

Figure 7: *Run times per iteration for varying granularity between polls, on a 16 processor Meiko CS-2.*

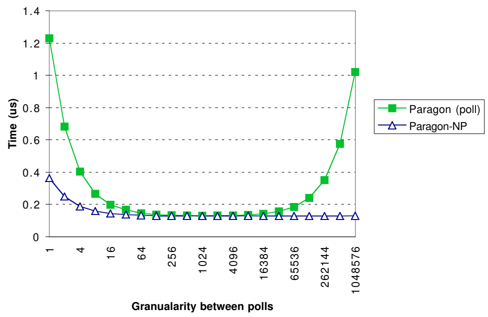

Figure 8: *Run times per iteration for varying granularity between polls, on an 8 processor Paragon.*

ularities, polling on the compute processor performs just as well as the NP implementation. Only when polling occurs very frequently does the polling overhead become noticeable.

5.4 Split-C programs

Finally, we compare the performance of full Split-C applications under the NP and Proc implementations on the Meiko and the Paragon. Table 2 lists our benchmark programs, along with the corresponding running times for the different versions of Split-C on the Meiko and Paragon. Figures 9 and 10 display the relative execution times as bar graphs. The programs were run on a 16 node Meiko CS-2 partition and an 8 node Paragon. Note that the problem sizes were different.

On the Meiko, we observe that under the NP strategy radix and sample run slower while mm, fft, shell, and wator run significantly faster. The remaining benchmarks generate similar timings. Radix and sample run slower because they are communication intensive and use remote events; radix uses stores to permute the data set on each pass while sample uses an atomic remote push to move data to its destination processor. These primitives are substantially slower under the NP implementation.

Most of the benchmarks have similar run times under the two implementations. This occurs for two reasons: First, most of these programs do not overlap communication and computation to a large degree. Instead, they run in phases separated by barriers. As a result, the NP cannot exploit its ability to handle communication while the compute processor computes. Second, much of the communication is done with bulk operations, and there is only a 20% difference in bandwidth between the two Split-C implementations.

The program with the largest improvement is wator. In this program each processor runs through a loop that reads a data point and then computes on that data. Since the Proc version only polls when it performs communication, any requests it receives while it is computing experience a long delay. The NP implementation, in contrast, can process requests immediately. This accounts for the large difference in run-times. To avoid at least some of this delay, the programmer would have to add polls to the compute routine. Unfortunately, this program invokes the X-library, the code for which is not readily accessible for inserting polls.

On the Paragon, almost all of the programs run approximately at the same rate under both the Proc and NP implementations of Split-C. The exceptions are the fine grained communication intensive programs — radix, sample, and fft — which run faster on the NP implementation because the underlying communication primitives are more efficient. The remaining exception is wator, which runs more efficiently under the NP implementation because of the improved responsiveness. As expected, programs that use bulk transfers do not show much change.

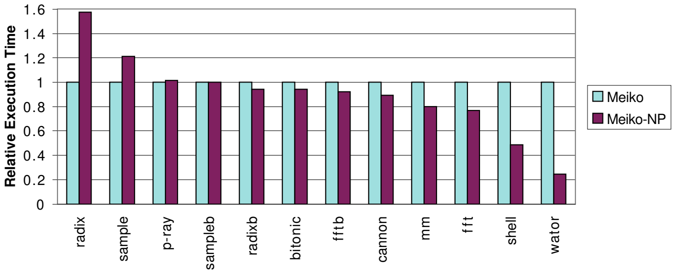

Figure 9: *Run times for our Split-C benchmark programs on a 16 processor Meiko CS-2, normalized to the running time of the main processor implementation.*

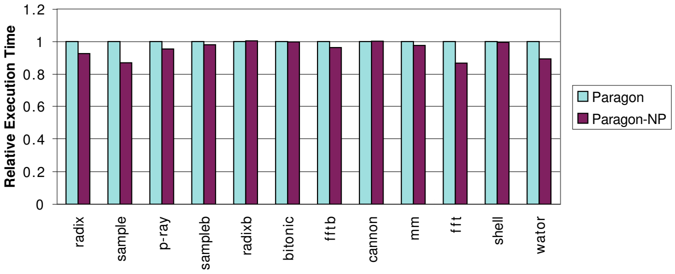

Figure 10: *Run times for our Split-C benchmark programs on an 8 processor Paragon, normalized to the running time of the main processor implementation.*

5.5 Discussion

The original motivation for an NP was to enable user-level protected communication without the limitations of gang-scheduling found on machines such as the CM-5. Protection is realized by running part of the operating system on the NP, either in software as on the Paragon, or in hardware as on the Meiko. Having the NP helps decrease the overhead observed on the compute processor, and thus may enable overlapping of computation and communication to a greater degree. On some of the machines, the NP also helps reduce the gap between messages and improve bulk transfers. An important advantage of the NP is that it improves the responsiveness.

There are several factors that prevent us from realizing the full utility of the NP, including limited speed, required protection, functionality of the NP and address translation, as well as the synchronization and the shared memory access cost between main processor and NP. For example, on the Meiko, the protection check, which involves table lookups, is expensive, since the NP does not have an on-chip cache. On the Paragon, the address translation has to be performed in software. Having the NP do the sending increases the observed latency as one more step is involved. Just getting the information from the compute processor to the NP is already quite expensive, since it involves an elaborate shared memory protocol. Finally, if the NP is a specially designed chip, as in the case of the Meiko and the NOW, it is likely to be much slower than the compute processor.

6 Conclusions

There has been a clear trend in the designs of large scale parallel machines towards more sophisticated hardware support for communication, better user-level messaging capabilities, and a greater emphasis on global address-based communication. In many cases, this has led in the direction of dedicated network processors; however, there is a great deal of variation in how specialized these are to the communication task, how they interface to the processor and the network, the kind of synchronization support they provide, and the level of protection they offer.

In this study we evaluate the tradeoffs present in this large design space by implementing a simple global address programming language, Split-C, on a range of these architectures and by pursuing a family of implementation strategies, each fully optimized for the capabilities of the hardware under that strategy. We see quite substantial differences in the latency, overhead, and gap exhibited on the individual global access primitives, and the differences are, in hindsight, readily explained. On most of our applications, the differences between the implementation strategies is less pronounced, partly because they tend to use primitives that were more uniform in performance across the strategies and partly because opposing trade-offs tend to balance out.

The experience of the study and the measurements that it offers provide some clear design guidelines for the communication substructure of very large parallel machines, as well as identifying points where the conclusion is still unclear:

- Imposing a network processor between the application program and the network provides a very simple (although not necessarily inexpensive) means of addressing the complex requirements of protection, address translation, media arbitration, and flow-control for communication. However, it is important that the interface between the processor and the network processor be efficient. This is not necessarily achieved by traditional bus-based cache-coherency protocols, since the idea is to move information from producer to consumer quickly, rather than to hold data close to the processor that touches it. It does seem to be achievable by a more specialized network processor integrated with the network interface.

- There is an advantage to having the network processor be responsive to the network and service memory access requests without waiting for the processor. However, if the network processor is going to do more than act as an intermediary and sanitize the network interface, it needs to be powerful enough to do this job with performance competitive with the processor. In particular, if the network processor is to provide a protection model powerful enough to allow its use on general purpose operations, it should be fast enough to be effective on those operations.

- The remote memory performance of the T3D and of certain aspects of the Meiko, show that there are clear benefits to be obtained through hardware support for specific operations in the network processor and network interface. However, the application usage characteristics on these large machines will need to stabilize before it will be possible to determine how these advantages balance against design time, cost, or reductions in performance elsewhere in the system.

Acknowledgments

We would like to thank Lok Liu, Rich Martin, Mitch Ferguson, and Paul Kolano for all of their assistance and expertise. We also thank Tom Anderson for providing us with valuable comments. This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contracts DABT63-92-C-0026 and F-30602-95-C-0136, by the Department of Energy under contract DE-FG03-94ER25206, by the National Science Foundation and California Micro Program. David Culler is supported by NSF PFF Award (CCR-9253705). Klaus Schauser is supported by NSF CAREER Award (CCR-9502661). Chris Scheiman is supported by NSF Postdoctoral Award (ASC-9504291). Computational resources were provided by the NSF Infrastructure Grants CDA-8722788 and CDA-9216202.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), 1996.

[2] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW(Network of Workstations). *IEEE Micro*, 15(1), February 1995.

[3] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *International Symposium on Computer Architecture*, June 1995.

[4] E. Barton, J. Cownie, and M. McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, 20(4), April 1994.

[5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Fifteenth ACM Symposium on Operating System Principles*, 1995.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1), February 1995.

[7] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with Olden (parallel programming). In *Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings*. Springer-Verlag, 1994.

[8] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 7(4), November 1989.

[9] K. M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. In *5th International Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[10] Cray Research Incorporated. *The CRAY T3D Hardware Reference Manual*, 1993.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Sumbramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 1993 Conference on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[12] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, February 1996.

[13] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Supercomputing '93*, Portland, Oregon, November 1993.

[14] W. Groscup. The Intel Paragon XP/S Supercomputer. In *Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology.*, Nov 1992.

[15] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0, May 1993.

[16] Kendall Square Research. *KSR1 Technical Summary*, 1992.

[17] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *7th ACM International Conference on Supercomputing*, July 1993.

[18] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford Flash Multiprocessor. In *21st International Symposium on Computer Architecture*, April 1994.

[19] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. The Network Architecture of the CM-5. In *Symposium on Parallel and Distributed Algorithms '92*, June 1992.

[20] D. Lenoski, J. Laundon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory Based Cache Coherance Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[21] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, November 1989.

[22] L. T. Liu and D. E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Intel Supercomputer Users Group Conference*, 1995.

[23] R. S. Nikhil. Cid: A Parallel, "Shared Memory" C for Distributed Memory Machines. In *Languages and Compilers for Parallel Computing. 7th International Workshop Proceedings*. Springer-Verlag, 1995.

[24] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Typhoon and Tempest: User-Level Shared Memory. In *International Symposium on Computer Architecture*, April 1994.

[25] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *9th International Parallel Processing Symposium*, April 1995.

[26] K. E. Schauser, C. J. Scheiman, J. M. Ferguson, and P. Z. Kolano. Exploiting the Capabilities of Communications Coprocessors. In *10th International Parallel Processing Symposium*, April 1996.

[27] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, 1992.

[28] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Fifteenth ACM Symposium on Operating System Principles*, December 1995.

[29] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, 1992.

[30] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Fourteenth ACM Symposium on Operating System Principles*, 1993.

[31] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for a Distributed Shared Memory. In *First Symposium on Operating Systems Design and Implementation*, 1994.