Appears in: Proceedings of the 22nd International Symposium on Computer Architecture, June 21–25, 1995, Santa Marguerita di Ligure, Italy

Optimizing Memory System Performance for Communication in Parallel Computers

T. Stricker[1] and T. Gross[1,2]

[1]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[2]Institut fuer Computer Systeme
ETH Zuerich
CH 8092 Zuerich, Switzerland

Abstract

Communication in a parallel system frequently involves moving data from the memory of one node to the memory of another; this is the standard communication model employed in message passing systems. Depending on the application, we observe a variety of patterns as part of communication steps, e.g., regular (i.e. blocks of data), strided, or irregular (indexed) memory accesses. The effective speed of these communication steps is determined by the network bandwidth *and* the memory bandwidth, and measurements on current parallel supercomputers indicate that the performance is limited by the *memory bandwidth* rather than the *network bandwidth*.

Current systems provide a wealth of options to perform communication, and a compiler or user is faced with the difficulty of finding the communication operations that best use the available memory and network bandwidth. This paper provides a framework to evaluate different solutions for inter-node communication and presents the *copy-transfer model*; this model captures the contributions of the memory system to inter-node communication. We demonstrate the usefulness of this simple model by applying it to two commercial parallel systems, the Cray T3D and the Intel Paragon.

In particular we identify two methods to transfer data between nodes in these two machines. In *buffer-packing* transfers, a contiguous block of data is transferred across the network. If the data are not stored contiguously, they are copied to (gathering) or from (scattering) buffers in local memory before and after the transfer. *Chained* transfers perform gathering, transfer and scattering in one step, reading the data elements with some non-sequential pattern and immediately transferring them on to the destination.

Our model and measurements indicate that chaining of the gather, transfer, and scatter operations results in better performance than buffer packing for many important access patterns. Most standard message passing libraries (like MPI, PVM or NX) force the parallelizing compiler (or the programmer) to employ the buffer-packing communication operations. However, the addition of hardware support dedicated to communication (e.g., DMAs, line-transfer units) now gives the compiler a wider range of options.

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152. Computational resources were provided in part by the Pittsburgh Supercomputing Center (PSC).

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

1 Introduction

Communication is a key issue for the design of a parallel computer, and the properties of the communication system have a high impact on the class of applications that profitably run on a parallel or distributed system.

Communication systems sometimes pay more attention to the network (i.e., the links, busses, or switches that connect the nodes in the parallel system) than to the suppliers and consumers of the data. Most communication steps in parallel systems involve moving data from the memory of one node to the memory of another node. The effective performance of the memory system is therefore (at least) as important as the performance of the communication system, and improving the network performance beyond what can be supported by the memory system does not increase overall performance.

The issue of transferring data between a node and its network is more complicated than just increasing the memory bandwidth. Although there is a clear trend towards increased memory bandwidth both in the nodes of parallel computers and in other systems (i.e., workstations), a large part of this performance improvement is due to widening the path between the processor and the memory. This change increases the memory bandwidth for contiguous (or almost contiguous) accesses, but does not increase the "reference bandwidth" (i.e., the number of references per instruction or cycle), nor does it improve the latency. Both of these aspects however are important if the data are accessed in some strided or irregular fashion.

The memory systems of modern parallel systems are complicated, and the performance of a sequence of accesses depends on a number of factors. Also, most parallel machines provide more than one way to implement the communication steps required by the program. Depending on the machine, there may be a choice of portable communication libraries (e.g., PVM or MPI), custom libraries, or low-level transfer operations like "put" and "get". A compiler or user is faced with a number of options, and it is not always easy to find the most efficient one.

This paper attempts to provide some answers to designers of the interface between the network and the memory/processor, as well as to compiler writers who want to custom-tailor a compiler's communication operations to a specific parallel system. We start with a brief review of communication in a parallel system to summarize the kind of data transfers that are required when applications are mapped onto modern parallel systems. Then we develop the *copy-transfer model* of inter-node communication; this model is simple enough to hide many details of the memory and communication systems, yet it allows us to characterize real parallel systems.

In the copy transfer model each communication step is written down exactly as it is carried out by the hardware. The formal description is *end to end* and must include all copies needed to gather and scatter the data, if buffered or non-contiguous accesses are involved. The model also captures whether a copy transfer is done in parallel or in sequence. As an example consider the transfer of a contiguous block of items that are then stored as a sequence with a constant stride of 64 on the remote system. If this operation (${}_1\mathrm{Tfer}_{64}$: starting stride 1, final stride 64) is implemented as a block transfer, followed by a copy to unpacking at the receiver, then this operation is written as:

$$ {}_1\mathrm{Tfer}_{64} = {}_1\mathrm{Network}_1 \circ {}_1\mathrm{Copy}_{64}. $$

Each basic transfer on the right side of our definition is associated with a measured throughput figure for a specific parallel system. The model contains a set of simple assumptions and rules to derive an estimated throughput for the transfer as described later.

After introducing a compiler view of communication in parallel systems and common memory access patterns, we define our model and show how to derive performance from measured basic transfer rates. We validate the model for two current parallel systems, the Cray T3D and the Intel Paragon and use it to evaluate two different ways to program communication operations. The methods based on our model (and confirmed by our experiments) are different from what is currently offered by the vendor software. We quantify the significance of our finding with measured performance of three common applications kernels.

2 Communication in parallel systems

In message passing systems, either the user or the compiler explicitly moves data from one node to another, thereby "renaming" the data. That is, as data are moved from one node to another, its name (address) is changed. In contrast, shared address space systems preserve the name of a data item as it is moved to another node. A data item may appear in the local memory (cache) of a node after a transfer, but its name (address) is still the same as it was before the transfer. The relative advantages of both machines have been discussed in numerous papers, and there exist a number of machines for either style. This paper concerns itself solely with message passing communication, because (1) any improvement to message passing communication helps current[1, 13, 3] and future machines[7] that provide this communication model, even if these machine support other models as well, (2) a number of commercial systems are based on message passing (including all systems with a large number of nodes), and (3) the hardware/software solutions offered for communication on these systems are far from satisfactory.

Modern message passing computers provide a variety of communication options, ranging from "get/put" or remote load and store to a traditional message passing interface (e.g., as encapsulated in libraries like PVM, NX, MPI, etc.). The communication styles found in these systems cover a wide range.

The hardware of message passing computers provides a high nominal communication bandwidth between nodes – the T3D has a hardware peak bandwidth of 300 MB/s on the wires between a pair of nodes, and the Paragon a peak hardware bandwidth of about 200 MB/s. In reality, control information (e.g., routing information, message delimiters) reduces this figure to about 160 MB/s for both machines. But even if we use a minimal protocol and a bare-bones runtime system, eliminate overhead through appropriate compiler technology, or handcraft the communication code, we do not observe even these measurable bandwidths for applications.

Figure 1 depicts the measured performance for PVM and low level libraries for the T3D and the Paragon. PVM provides buffered message passing with general send/receive semantics, while the lower level primitives in the vendor specific libraries libsma.a on the Cray and libnx.a on the Paragon (SUNMOS) allow fastest transfers with semantic restrictions, such as executing receives before the sends or relying on user managed cache consistency.

Experimental studies of actual applications indicate that the effective communication throughput never reaches peak bandwidth, even if applications are scaled to giant problem sizes. After a careful examination of overheads, we find that it is not the constant per message overhead due to the operating or runtime system that is to blame (if this was the cause of our problems, we would observe a steady performance increase as we scale the size of the benchmarks sets), but rather overheads that occur for each byte transferred.

All data transfers start and end in memory. So the performance of the memory system for communication plays a crucial role in determining the overall performance of applications running on parallel machines. We observed that there are many applications for which the difficult, non-sequential memory access patterns occur mostly in connection with communication. For example, when mapping a 2D FFT (consisting of 1D FFTs and a transpose) onto a parallel computer, the 1D FFTs can be organized to run with locality out of caches, and the memory accesses without locality are part of the transpose. Since the performance of the memory system is so important for communication, we now turn our attention to the generator of the communication operations, the compiler.

2.1 Compiler view of communication

To map an application onto a parallel system, the compiler must determine how data and computations are to be distributed over the nodes of the parallel system. Recently, the High Performance Fortran (HPF) effort has resulted in a set of *user directives* that assist the compiler in performing its tasks[5][1]. HPF focuses on block-cyclic distribution of arrays, where the two variants, the block and cyclic are the most common [15]. The distributions included in standard HPF are well-suited to describe regular data layouts. However, many applications are *irregular* in that the access pattern cannot be described with a few parameters. Instead, the access pattern is contained in another data structure, usually referred to as an index array. A typical example is `A[1:n] = B[X[1:n]]` where `X` contains some permutation of `1..n` (i.e., there are no duplicate entries in `X`). A great deal of compiler effort is required to deal with the complexities of such code; after all, `A`, `B`, and `X` might all be distributed over multiple nodes. However, the bottom line is that the compiler at some time has to access the elements of `B`, using some intermediate index array `T`, as depicted in Figure 2.

From a compiler's point of view, data are moved between the address spaces of nodes, and these data can be contiguous blocks, slices, intersections of slices[15], or irregular blocks of

[1] Our work is done in collaboration with the implementation of an HPF compiler[6], but the details of HPF are irrelevant to this study. Our results apply to any system that moves data from the local memory of one node to the remote memory of another.

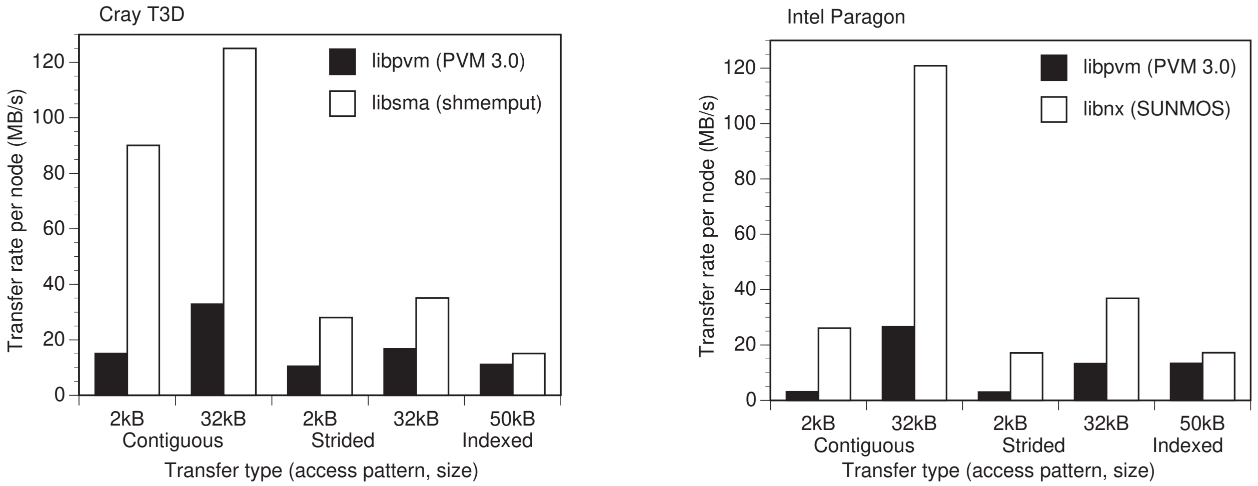


Figure 1: Measured application throughput for simple communication operations with a portable, general library (PVM) and with vendor specific or third party libraries that offer best throughput.

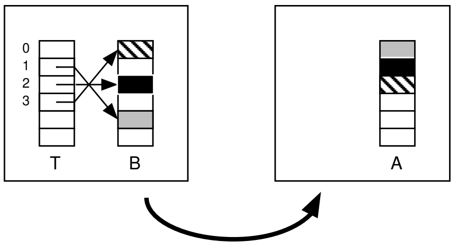


Figure 2: Access with an index array.

data described by an index array. The compiler generates synchronization (or control) instructions separately (e.g., before and after a complete array redistribution) [16]. This organization allows us to focus in this paper on speeding up the *data transfers*. There are two principal approaches to organizing the data transfers. Either the compiler invokes communication operations as provided by a conventional message passing library (and posts all receives before starting the send operations to streamline processing of incoming messages). Or the compiler uses remote stores to "put" the data to their destination. It can generate the addresses for the loads and stores on either node, the sender node or the receiver node.

2.2 Memory reference patterns

The code generated by the compiler for each node to transfer data attempts to take a number of factors into account: the specific distribution, the size of the array (if known), and the size of the parallel system (if known). From the perspective of the memory system of a node, we can observe three different types of memory access in support of communication:

Contiguous The memory access to is a contiguous block of data. Our basic unit of transfer is a 64bit word, often a double precision floating point number. This pattern commonly results from *block distributions*.

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

Indexed An arbitrary sequence of words is accessed. The specific array access pattern is determined by indices given in a separate index array. Reading the indices is overhead; reading the index is considered to be part of the memory access operation and does *not* count towards what we report as the effective memory access bandwidth for an application. Indexed patterns are common for irregular distributions and sparse matrix representations[14].

Although strided accesses are often the consequence of a cyclic or block-cyclic distribution, it is also possible that they result from a blocked distribution.

2.3 Memory systems of parallel computers

For this paper we present a simplified view of the node architecture that focuses attention on the basic architectural components relevant to our parallel compilation model. We assume a basic local memory system with a primary cache in the microprocessor and a DRAM-based memory system. We also assume that data are sent and received through a simple transfer to the network interface (e.g. load/store to a FIFO). For the model it is important to capture parallel operation of additional functional units capable of doing memory operations such as DMA controllers or fetch/deposit engines that process incoming get and put requests without the involvement of a processor. In Section 3.5 these general concepts are related to actual hardware in the Cray T3D and Intel Paragon.

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

Even a simplified view of the memory system allows for a rich set of choices for a compiler to organize the inter-node communication. The objective of a compiler is to obtain highest possible communication performance for transfers with the communication and memory access patterns required by parallel programs. In this section we introduce a model to reason about different sequences of operations involved in such data transfers. This model can be used to estimate the maximal transfer performance (throughput) as well as to determine rules for generating the best code by a parallel compiler.

3.1 A throughput-oriented model

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

In general the performance of cached memory systems cannot be specified by memory access bandwidth and latency alone. The memory system performance critically depends on temporal locality. Traditionally the need to accurately analyze the memory system performance for compilers lead to trace driven investigations of the cached memory system. In summary, operand reuse and temporal locality work well to improve the performance of computation if blocked algorithms and optimized kernels (like BLAS3) are used. However, we observe that temporal locality plays only a small role in the memory accesses for communication. We devise a *throughput* oriented model, that is easier to use for a compiler writer than memory access traces, and that nevertheless reflects the performance experienced by applications.

The importance of throughput is not surprising given the properties of communication related memory accesses. In data parallel programs, parallelism is exploited by operation on large collections, with the data distributed over a large number of processors[2]. In practice, these collections can be quite large and a compiler cannot assume that the local data structure on any node fits entirely into the local cache of a node.

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

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

3.2 Basic transfers

All compiler-generated communication operations can be decomposed into basic transfers or steps. We now introduce some terminology to capture the key aspects of these basic steps, which concentrate on common access patterns encountered in parallelizing compilers. A transfer T moves data using a source pattern r and a destination pattern w. The source and destination patterns capture the memory access patterns, i.e. how the data are read and written. The read (load) and write (store) locations are always on the *same* node, unless explicitly noted. To concisely represent such a step T, we mark the read pattern as a *left* subscript and the write pattern as a *right* subscript, i.e. ${}_rT_w$. Typical patterns are 1 for contiguous accesses, $2, 3, \ldots$ for strided access with constant strides of $2, 3, \ldots$, and ω for indexed accesses. We use the access pattern 0 if the source or destination is a fixed location in memory (e.g., the head or tail of a FIFO) .

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

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

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

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

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

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

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

N_d **data-only network** The N_d transfer moves only data across the network.

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

3.3 Estimating throughput for communication operations

We can now compose communication operations for a variety of access patterns by concatenating basic transfers. We establish two concatenation rules and operators: Two transfers using the same resources (e.g., the processor) must be concatenated in sequence $\circ$. The write (left subscript) access pattern of the first transfer must match the read (right subscript) access patterns of the second transfer. Transfers that use disjoint communication resources can occur in parallel $\|$.

The formal description of the communication operations as basic transfers allows us to estimate the maximal transfer throughput for several implementations of a communication operation. We use the following three rules to derive an estimate for the effective throughput $|Z|$ of a communication operation Z based on the throughput of the basic transfers involved.

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

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

$<$ **Resource constraint** In performance estimates the model can consider additional resource constraints to limit the total throughput of certain transfers that can occur in parallel. For example, if the processor and the DMAs share a common system bus, the total bus bandwidth cannot be exceeded. Resource constraints are given as inequality of bandwidth parameters. If a resource constraint cannot be met, the throughput parameter of the participating basic transfers must be reduced until the constraint is met.

3.4 Example: Buffer-packing transfers, PVM style

The performance critical communication operation used by the communication code of a parallel compiler is a local memory to remote memory copy ${}_xQ_y$. Depending on the distributions of the array operands of an array assignment, different access pattern may be encountered for load accesses (x) at the source and store accesses (y) at the destination.

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

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

It might appear that for contiguous transfers $({}_1Q_1)$ the first and the last memory copy $({}_1C_1)$ are unnecessary. This is true in principle, but message passing libraries like PVM force the programmer/compiler writer to copy the data elements in all cases to comply with the standard application programming interface. Of course, there may be different ways to implement ${}_xQ_y$, especially if constraints are placed onto x and y, and we return to this topic in Section 5. But first we discuss how to obtain the throughput figure of interest for communication operations composed by the compiler from the basic transfers.

3.4.1 Throughput of buffer-packing transfers

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

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

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

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

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

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

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

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

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

3.5 Architecture support for communication operations

We briefly review the Paragon and the T3D. We take the liberty to omit the description of those parts that are irrelevant for our study, e.g. the support for remote loads, fetch and increment, or atomic swaps on the T3D, or that are not supported by the current software system and are therefore not accessible to any application or measurement tool. We refer the interested reader to the reference literature about these machines for further technical details[1, 3, 12].

The compiler demands communication with transfers ${}_xQ_y$ for all access patterns x and y, including strided and indexed. To move data from one node to another, several parallel systems include some form of hardware support to "drop" or "deposit" the data into the memory of the destination node. This hardware may also be usable to "pull" or "withdraw" data from the memory of the source node, but we emphasize the deposit aspect since we observed higher performance in practice. [2] We refer to the hardware support for receiving remote stores as a *deposit engine*. The sole purpose of a deposit engine is to take data from the network and store it to the memory system on behalf of the communication system. It is important that these transfers take place automatically, without further node involvement, i.e. in the background of whatever computation or communication takes place on the node. This requirement to operate concurrently with send operations distinguishes deposit engines from handlers, as found in software solutions like active messages[17]. Handlers attempts to provide a solution to a more general problem; their invocation may involve a control transfer or even crossing of protection boundaries (e.g. as part of an RPC). In contrast, a deposit engine is geared towards a single task, remote stores and can perform this task independently, in parallel and efficiently at the full speed of the network.

[2] Briefly stated, the reason for this is that when depositing data, address information and data travel once together over the network. When withdrawing data, the latency is higher since address information has to travel first to the node that holds the data.

3.5.1 Cray T3D

A T3D node consists of a 64bit DEC Alpha microprocessor, a local memory system, a memory mapped network interface to send remote stores to the network, and a deposit engine called the annex. The memory of a T3D node is a simple non-interleaved memory system built from DRAM chips. Unlike workstations, the node has no virtual memory.

The interface between the computation agent and the main memory is an 8KB primary cache, which is implemented on-chip within the Alpha microprocessor. The memory system and its interface to processor and communication are shown in Figure 3. External read-ahead circuitry (RDAL) can be turned on by the programmer at load-time to improve performance of contiguous load streams; we have measured improvements of approx. 60%. For writes, the default configuration of the cache is write-around, and support for writes consists of the write back queue (WBQ) provided by the microprocessor. The documentation of the Cray T3D Application Programmers Course [4] specifies the local read bandwidth at 55 MB/s for non-contiguous single word transfers, and up to 320 MB/s for contiguous reading of cache lines with read-ahead. The latency of a load from main memory is around 150ns.

The interface between the processor and communication system on the Cray T3D consists of the annex, a memory mapped communication port, which maps some range of free address space to the physical memory of another node in the system; this node is then selected as a communication partner. The communication partner can be switched with a fixed overhead by modifying the appropriate annex entry. Once a store operation is issued to the communication port, the communication subsystem takes over the specified address and data, and it sends a message out to the receiver. Remote loads are handled in a similar way.

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

Transfers from the processor to the communication system can be performed at a rate of approximately 125 MB/s, and if multiple nodes perform remote stores of contiguous blocks to a single node, these transfers can be processed at the full network speed (160 MB/s)[12].

3.5.2 Intel Paragon

The node of a Paragon system contains multiple processors sharing a common memory. Our investigation is based on a system with 2 processors/node, but systems with 3 processors/node have been built as well. Except for the mechanisms to support multiple processors, the memory system of the Paragon system is surprisingly similar to the Cray T3D. The memory system and its interface to processor and communication are depicted in Figure 3.

The processors of a Paragon node are two Intel i860XP processors. Both processors have their own primary on-chip data cache and are connected to the local memory system over a 400 MB/s high speed bus. The data cache is 16 KB, organized 4-way associative, write-back or write-through. Under SUNMOS [10] (the operating system of choice for low-latency communication) the caches are write through. The i860XP processors contain support for higher bandwidth through pipelined loads (using the PFQ) that bypass the caches.

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

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

4 Measuring throughput figures for basic transfers

Although the detailed mechanics of the architectural support for each basic transfer are quite complex for each system, the performance can be measured in simple experiments using fine grain timers. These measurements result in a throughput figure for every basic transfers of Section 3.2.

The following tables and figures give the key bandwidth and throughput parameters for the T3D and the Paragon, measured on "live" systems in real time. The measurements of the effective bandwidth for the basic transfers are highly accurate and consistently reproducible. The basic transfers are defined in such a way that the throughput is based on the array elements transferred, and auxiliary data like headers, addresses, and even index loads are factored into the throughput figure. That is, these operations, although possibly consuming "raw" bandwidth, do not contribute to the net bandwidth an application is interested in. The model is optimistic in terms of interleaving the instructions and accesses of all basic transfers within a node and its memory system. It is assumed the usage of processor and memory system is spread evenly, over the duration of the whole communication operation. In practice, this is often obtained through pipelining.

4.1 Throughput of local copies

The throughput for the basic local memory-to-memory transfers ${}_xC_y$ critically depends an the access pattern as seen in Table 1.

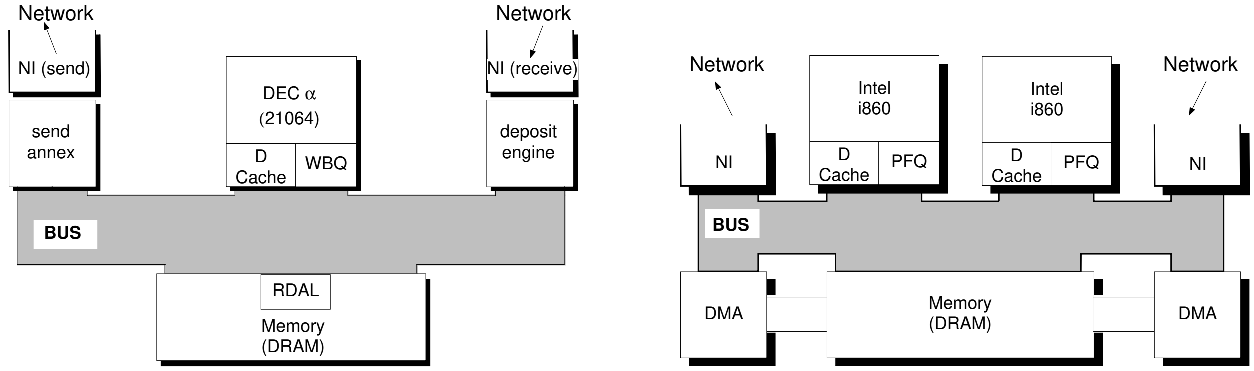
	$\lvert {}_1C_1 \rvert$	$\lvert {}_1C_{64} \rvert$	$\lvert {}_{64}C_1 \rvert$	$\lvert {}_1C_\omega \rvert$	$\lvert {}_\omega C_1 \rvert$
T3D	93	67.9	33.3	38.5	32.9
Paragon	67.6	27.6	31.1	35.2	45.1

Table 1: Throughput of selected local memory-to-memory transfers (MB/s) for large blocks.

The graph in Figure 4 show the different characteristics of the memory systems on T3D and Paragon, when strides are involved. On the T3D strided stores are better supported because of the write back queue. On the Paragon strided loads can be pipelined and benefit from the pre-fetch queue.

4.2 Throughput of send/receive copies

The throughput for the network access depends partly on local memory-to-memory transfer and partly on network limitations. The measured figures are given in Tables 2 and 3. Since the numbers do not vary for large strides, we assume for simplicity that the throughput for stride 64 applies to any larger stride.



a) b)

Figure 3: Overview of the two node architectures: a) T3D and b) Paragon. *NI* refers to the network interface chips and FIFOs.

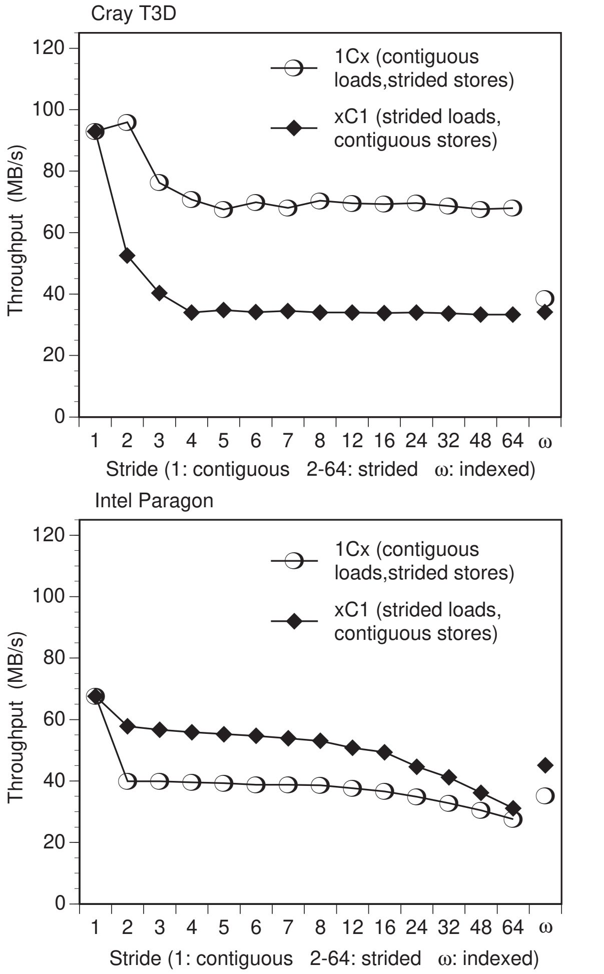


Figure 4: Throughput for strided local memory-to-memory transfers (MB/s).

	$\lvert_1 S_0\rvert$	$\lvert_1 F_0\rvert$	$\lvert_{64} S_0\rvert$	$\lvert_\omega S_0\rvert$
T3D	126	-	35	32
Paragon	52	160	42	36

Table 2: Throughput figures for sending network transfers (MB/s).

4.3 Congestion and throughput of the network

Network congestion is *absent* from our model. This may seem surprising at first, since none of the machines of interest to us

	$\lvert_0 R_1\rvert$	$\lvert_0 D_1\rvert$	$\lvert_0 R_{64}\rvert$	$\lvert_0 D_{64}\rvert$	$\lvert_0 R_\omega\rvert$	$\lvert_0 D_\omega\rvert$
T3D	-	142	-	52	-	52
Paragon	82	160	38	-	42	-

Table 3: Throughput figures for receiving network transfers (MB/s).

provides a fully scalable bisection bandwidth as e.g. the CM-5[9]. Both machines use a simple mesh topology with fast links for their communication networks. In our experience, the raw link speed in the network significantly exceeds the effective throughput achievable in useful data transfers. For most applications, the machines will not be network-congestion limited unless we move to very large machines. There are however two quirks: On the T3D, two adjacent nodes share a single communication port to the network. This design feature introduces congestion at the access point, and therefore the minimal congestion is *two* unless half of the processors remain unused. For the Paragon, the unfortunate aspect ratio of certain machine sizes (e.g., 112x16) and the lack of torus links can cause congestion for some patterns. In general, next neighbor patterns like cyclic shifts cause just a small congestion of one or two, and even dense patterns like the complete exchange or personalized all-to-all communication can be scheduled with minimal congestion on T3D tori of up to 1024 (2x8x8x8) compute nodes[8].

Because of these two problems in the T3D and Paragon networks, communication runs at a congestion of two in many cases, and we use the measured throughput for this congestion, when using our model to compute overall throughputs. For completeness, Table 4 shows network performance at congestion one, two, and four. Congestion two means a network link is traversed by twice as much data as it can support at peak speed. For a throughput oriented model it is irrelevant whether the data are multiplexed at a per flit or a per message level.

For the network throughput, it is more important whether just the data words are transferred, or if the addresses for remote stores are transferred along with the data words (address-data pairs). We have therefore measured the network bandwidth for large block transfers for both options (data only and address-data pairs) for different fixed congestion factors. The bold data in Table 4 indicate what we consider to be representative values for our class of applications.

5 Optimization of communication operations

The large variety of access patterns and hardware capabilities implies that there are different ways to implement a particular

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

Table 4: Network bandwidth (MB/s) as a function of a fixed overall congestion.

communication operation ${}_xQ_y$ by composing it out of different basic transfers. Looking at the Cray T3D and the Intel Paragon, we identify different tradeoffs in the design of the most important communication operations of a parallelizing compiler. In both cases the copy-transfer model guides an optimization towards maximal performance.

5.1 Buffer-packing vs. chained transfers

Section 3 presents an example of buffer packing, but with appropriate hardware support, the buffer packing/unpacking copy steps can be eliminated. That is, we can implement the communication operations ${}_xQ_y$ for the T3D and the Paragon so that they avoid packing buffer(s). These implementations (and their bandwidths) are different for the two machines, but the overall idea is the same. Therefore, a compiler or user has two options when selecting communication operations to perform a computation step:

Buffer-packing transfers The buffer packing message passing libraries (such as PVM) attempt to transfer contiguous blocks at all costs, leaving the packing / unpacking of communication buffers to the application code. Packing and unpacking is done through a local copy in memory before and after the transfer across the network. This arrangement benefits from *faster* contiguous transfers across the network but suffers from the cost of additional accesses to local memory. Figure 5 illustrates the path of data for this style (of course, these operations are overlapped or pipelined as stated in Section 3.4).

Chained transfers By chaining the *slower* non-contiguous accesses to data with the transfer of data from local memory to the network at the sender side (and vice versa for the receiver side), we eliminate local copies at the expense of supplying the data more slowly to the network. The chained transfers rely on the deposit engine at the receiver node to perform the stores. Figure 6 illustrates the flow of data within a node.

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

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

5.1.1 Buffer packing transfers on the T3D

In the previous section, we presented the formula for buffer-packing message passing. This message passing style is provided by both the Cray PVM library on a higher level and the Cray SH_MEMPUT library (libsma.a) on a lower level. While both libraries contain primitives for direct contiguous memory transfers, both libraries fail to provide adequate direct transfers for strided and indexed transfers without local copying in memory. Furthermore, the performance of PVM is affected by additional copies to temporary system buffers

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

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

Using the model of Section 3, we obtain these performance estimates:

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

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

5.1.2 Chained transfers on the T3D

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

$${}_1Q'_1 \;=\; {}_1S_0 \| N_d \| {}_0D_1$$
$${}_xQ'_y \;=\; {}_xS_0 \| N_{adp} \| {}_0D_y$$

Using the concatenation rules of Section 3.3, our model predicts:

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

Figure 7 shows measured throughput rates for buffer packing and chained transfers on the T3D, for different access patterns. As can be seen, the model predictions match fairly accurately the measured performance.

5.1.3 Buffer packing transfers on the Intel Paragon

On the Paragon, the realization of different implementations ${}_xQ_y$ for buffer packing and ${}_xQ'_y$ for chained communication are less evident. At first sight, the Paragon, and many other conventional message passing architectures, appear to support only transfers of contiguous blocks over the network. So for a read and write pattern of 1, we can use the DMA as a deposit engine, but for other patterns, we have to fall back to buffer packing.

$$\begin{array}{lcl} {}_1Q_1 & = & {}_1F_0 \| N_d \| {}_0D_1 \\ {}_xQ_y & = & {}_xC_1 \circ ({}_1F_0 \| N_d \| {}_0D_1) \circ_1 C_y \\ (2 \times_x Q_y) & \leq & {}_0C_1 \text{ and } \leq {}_1C_0 \end{array}$$

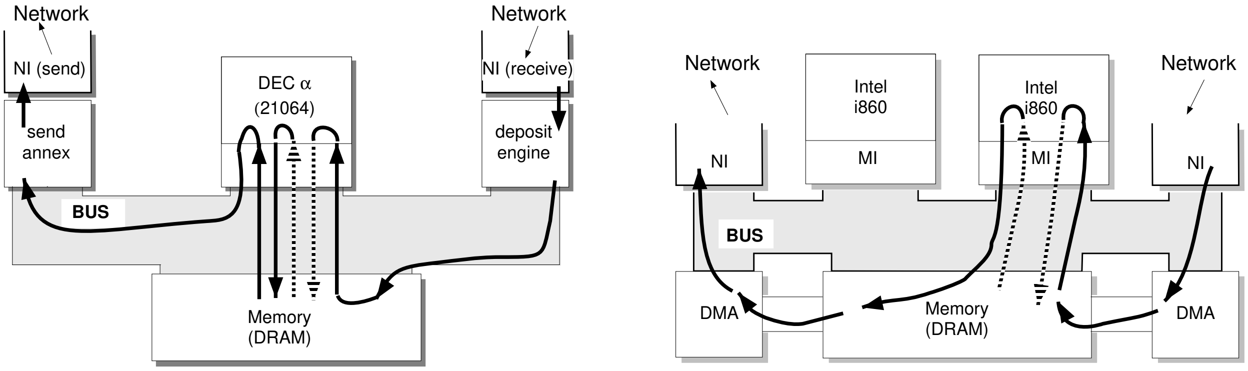


Figure 5: Schematic flow of data for buffer packing communication. Solid lines indicate streams of contiguous data, dashed lines potentially strided or indexed data.

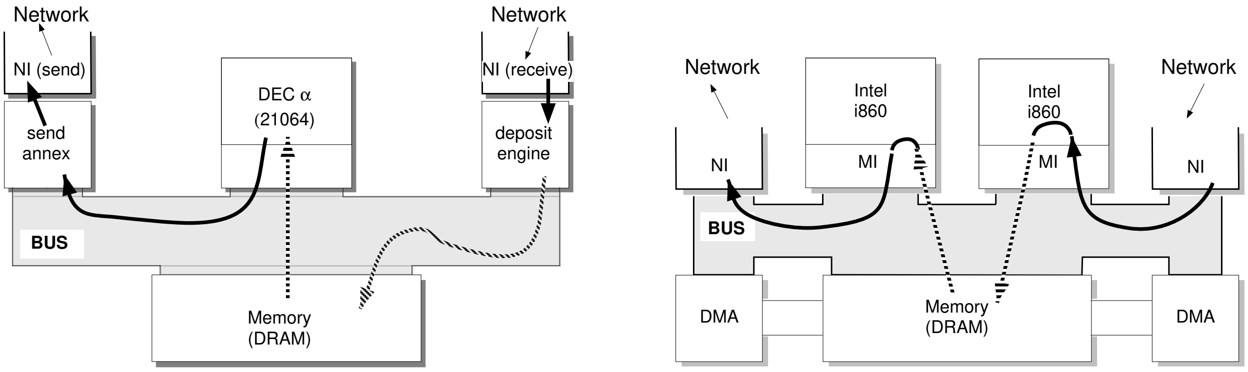


Figure 6: Schematic flow of data for chained communication. Solid lines indicate streams of contiguous data, dashed lines potentially strided or indexed data.

The data are first gathered in a memory-to-memory copy, thereafter it is transfered to the network with a send copy. At the receiver node, the data are stored in a contiguous buffer before it is scattered with a memory to memory copy. It is important that the contiguous transfers to the network are performed by DMAs and therefore they can be partially overlapped. Still, in practice the DMA engines on the Paragon require permanent attention of a processor; they need to be "kicked" back on if they stall either due to crossing a memory page boundary or due to hardware bugs in the communication interface chips. A full overlap with buffer packing occurs when the separate communication processor takes care of attending to the DMA engines, as is done in OSF/1 and mode 1 (co-processor reserved for communication) of SUNMOS:

$$\begin{array}{lcl} {}_xQ_y & = & {}_xC_1 \circ ({}_1F_0 \| N_d \| {}_0D_1) \| {}_1C_y \\ (2 \times_x Q_y) & \leq & {}_0C_1 \text{ and } \leq {}_1C_0 \end{array}$$

For these transfers, the model predicts:

$$\begin{array}{ll} |{}_1Q_1| = 20.7 \text{ MB/s} & |{}_1Q_{64}| = 16.1 \text{ MB/s} \\ |{}_{16}Q_{64}| = 14.9 \text{ MB/s} & |{}_\omega Q_\omega| = 16.2 \text{ MB/s} \end{array}$$

5.1.4 Chained transfers on the Intel Paragon

An efficient implementation of the chained communication primitive ${}_xQ'_y$ for arbitrary patterns x and y critically depends on the capabilities of the deposit engine. The current Paragon nodes provide only an inflexible DMA engine, which handles only contiguous accesses, has too many alignment constraints, and cannot even work across DRAM page boundaries. This DMA engine therefore does not meet the requirements for strided and indexed transfers.

A closer look at the node architecture in Figure 3 points towards a possible solution. The communication co-processor can be used *exclusively* as a deposit engine during communication. With a communication processor at work, remote stores can be implemented without disturbing other activities; any send operation can be done by the main processor. With this change, we obtain parallel execution of the basic transfer steps:

$$\begin{array}{lcl} {}_1Q'_1 & = & {}_1S_0 \| N_d \| {}_0R_1 \\ {}_xQ'_y & = & {}_xS_0 \| N_{adp} \| {}_0R_y \end{array}$$

In this case, the model estimates this performance:

$$\begin{array}{ll} |{}_1Q'_1| = 52\text{MB/s} & |{}_1Q'_{64}| = 38\text{MB/s} \\ |{}_{16}Q'_{64}| = 38\text{MB/s} & |{}_\omega Q'_\omega| = 36\text{MB/s} \end{array}$$

The model numbers tell us that if it is indeed possible to use the processor and co-processor simultaneously for memory accesses, the chained model could be a winner. The co-processor easily performs the task of a deposit engine. The major caveat comes from resource constraints in the model. If there is, e.g., a heavy penalty for bus arbitration between processor or co-processor, the second processor would be unable to help with communication work involving memory accesses. Only with the DMA can the data move over the network at full speed. This is an advantage on machines with a network that cannot share the bandwidth of a physical link among multiple virtual channels by multiplexing.

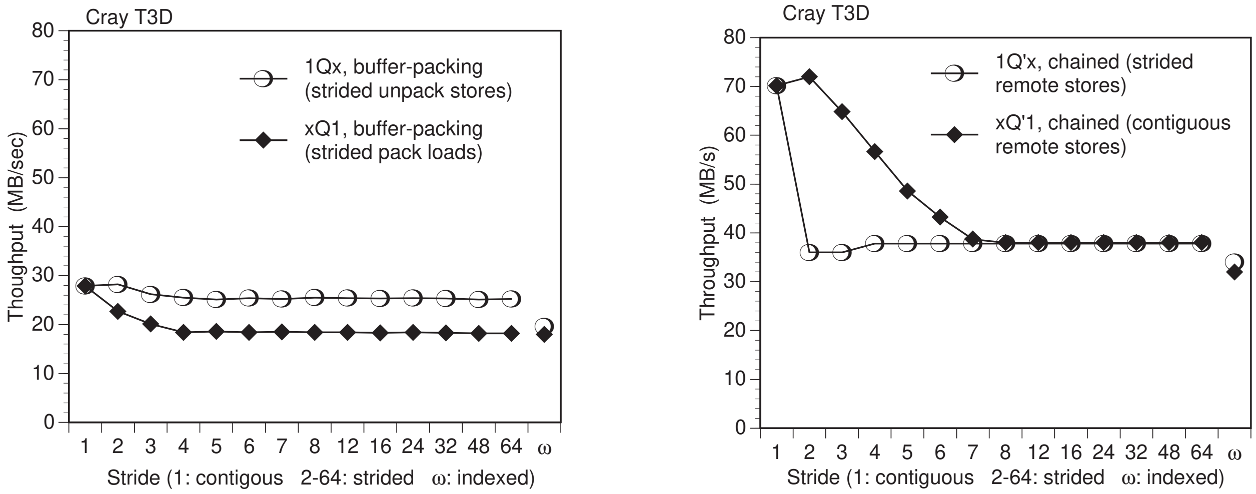


Figure 7: Throughput for communication operation with different strided access patterns including contiguous, strided and indexed for either loads or stores. The buffer-packing implementations (left) result in a lower throughput than the chained implementations (right).

Figure 8 shows measured throughput rates for buffer packing and chained transfers on the Paragon. However, due to difficulties with our buggy A-step network interface parts, the measurements deviate significantly from our conceptual model since we were (1) unable to use the pipelined loads (a 30-40% performance loss) and (2) we did not run sending and receiving simultaneously at each node. Experiments with simultaneous, interleaved memory accesses of processor and co-processor indicate that the bus in the current Paragon systems is not equipped for fine grain interleaving of single word loads and stores, and that a performance penalty of up to 50% must be expected.

5.2 Strided loads vs. strided stores

When implementing the communication primitive for a two dimensional array transpose, the compiler can choose between an access pattern of ${}_1Q_n$ or ${}_nQ_1$ in the remote memory transfer, as seen in Figure 9.

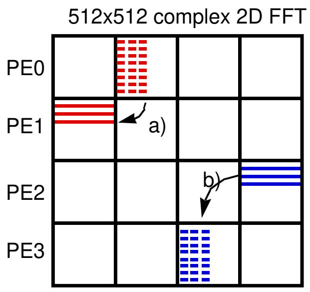


Figure 9: Execution of a 2D FFT includes an array transpose to change the distribution from row-major into column-major. Square patches must be moved between the processors. The patches of data can be moved in two ways, a) or b).

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

Using the bandwidth parameter rules of our copy-transfer model, the effective bandwidth of the communication operations is predicted as seen in the Table 5.

This optimization of choosing strided stores on the T3D and strided loads on the Paragon is not surprising, given the better performance of strided stores for memory-to-memory copies in one architecture and strided loads in the other architecture. We found that both the write back queue of the T3D and the prefetch queue on the DEC Alpha as well as the pipelined loads of the Intel i860 improve communication performance, especially for complex indexed pattern. Unfortunately, the standard single-node compilers do not generate code for these instructions.

6 Measured performance in application kernels

To evaluate the appropriateness of the copy-transfer model for applications (and not just basic communication operations as discussed in Section 5), we choose the communication kernels of three important applications. Two of these applications are compiled by a compiler for our dialect of HPF and one by an application-specific compiler. They are run on the T3D (since it is easier for us to explore architectural aspects on this machine than on the Paragon). The three applications were chosen to observe representative communication patterns.

6.1 Application kernels

The three kernels we used for our evaluation are: an array transpose, as it occurs in 2D FFT, the communication of a solver step in a finite element method (FEM) program and the communication occuring in a successive over-relaxation (SOR) solver.

6.1.1 Transpose in 2D FFT

Transposes are important to many application. Our example is taken from an $n \times n$ 2D FFT application kernel. We choose a 1024×1024 complex 2D FFT because we observed this problem size to be common for applications on this class of machines. The transposes are necessary to provide locality for the column FFTs after the row FFTs are completed. We encountered a transpose of similar size as the performance critical communication step of a grand challenge application in air-shed modeling [11]. This code redistributes a $3500 \times (35 \times 5)$ array

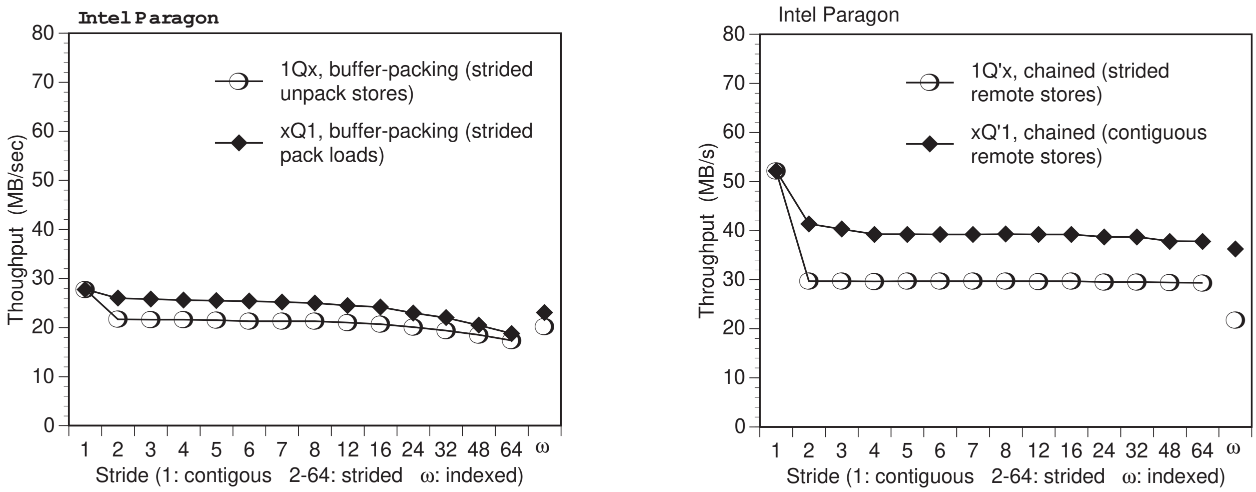


Figure 8: Throughput for communication operation with different strided access patterns including contiguous, strided and indexed for either loads or stores. The buffer-packing implementations (left) result in a lower throughput than the chained implementations (right).

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

Table 5: Estimated and measured performance for strided loads vs. strided stores.

between one phase that performs numerical chemistry calculations and another phase that calculates transport phenomena, and this redistribution is implemented as a generic transpose.

6.1.2 Iterative solver on partitioned Finite Element graph

The FEM application kernel is derived from a sparse system solver based on a partitioned finite element graph, representing a 3 dimensional model of an alluvial valley surrounded by hard rock. This graph is used by our colleagues to study earthquakes [14]. Since the structure is an irregular well partitioned grid, only a fraction of the local data elements is exchanged between nodes, and the communication involves indexed accesses with arbitrary strides.

6.1.3 Successive over-relaxation solver

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

6.2 Measurements

For each application kernel we determine the throughput of the communication step for both buffer-packing communication and chained communication. Table 6 shows the throughput estimate of our model as well as the actual measurement on a 64-node partition of a T3D.

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

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

Table 6: Measured data transfer rates of our application on a 64-node partition of a 512-node T3D, (MB/s *per node*).

7 Conclusions

Parallel supercomputers provide a high raw communication bandwidth, but applications realize only a fraction of the stated peak bandwidths. Since the data to be transferred from one node to another are moved from the memory of one node to the memory of another node, we follow the path of data through the system and discover that the memory system performance is actually the limiting factor for many applications. Particularly applications that move strided blocks, or use an index array to look up the elements to be transferred are susceptible to memory system performance.

Modern parallel systems are complex and therefore pose a challenge to any compiler writer who wants to keep down the overhead cost of communication. To assist compiler writers, we developed the copy-transfer model to allow tradeoffs between different implementation strategies for communication operations. This model is driven by throughput figures for the three different memory access patterns generated by compilers, as well as by a few key performance parameters of the communication network. We applied the model to two current parallel systems, the Intel Paragon and the Cray T3D to analyze both basic communication operations as well as the kernels of some key applications. Although simple, the model is highly accurate in the cases that we have evaluated so far.

Improving the performance of the memory system at each processing node is not feasible on installed machines, and design changes may or may not be economical for a massively parallel system. We therefore focus our conclusion on software options for optimizing the memory performance of communication operations.

Depending on the details of the memory system and the addressing pattern of the application, it may be more advantageous to transfer the data directly from their home location, without first compacting them into a contiguous block of memory. We call this "chained" communication and relate it to "buffer-packing" communication, as it is done by many conventional message passing system. The insight that "chained" communication can perform better was first demonstrated by the our simple model and then verified in practice for two modern parallel systems, the Cray T3D and the Intel Paragon. For three relevant application kernels, these tests confirm that "chained" communication results in 40-60% higher performance for access pattern other than contiguous accesses on the Cray T3D.

"Chained" communication relies on the design of the *deposit engine* (e.g., block transfer engines, line transfer units, or DMAs) to handle receiving the data *in the background*. Additional hardware support is only useful to the extent that it supports the demands of a parallelizing compiler. That is, such engines must take into account that not all transfers are contiguous blocks of compact data. Furthermore, engines that have a large unit of transfer (say more than 4 operands, or even pages) may not deliver the expected performance in application, because the transfer will be limited by memory access necessary to prepare the communication operation.

The crucial role of memory system performance is not novel to the supercomputing world. As has been observed in studies of vector-supercomputers, it is often the memory system that makes or breaks an application. The same holds true for parallel systems. The parallelism exploited in applications is no panacea and cannot cover up inadequate memory system performance. To the contrary, as the interconnect bandwidths and latencies of parallel computers improve, the demands on the memory system are going to increase. We observed the utility of write back buffers and pipelined loads, contributing to better memory performance. It is important that the designers of such systems pay attention to the memory system demands of parallelizing compilers if they want to build a hospitable platform for applications.

References

[1] D. Adams. Cray T3D System Architecture Overview. Technical report, Cray Research Inc., September 1993. Revision 1.C.

[2] G. Blelloch and J. Sipelstein. Collection-Oriented Languages. *Proc. IEEE*, 79(4):504–523, Apr 1991.

[3] Intel Corp. *Paragon X/PS Product Overview*. Intel Corp., March 1991.

[4] Cray Research Inc. *CRAY T3D Applications Programming Course*, Nov 1993. TR-T3DAPPL.

[5] High Performance Fortran Forum. High Performance Fortran language specification version 1.0 draft, January 1993.

[6] T. Gross, D. O'Hallaron, and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.

[7] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata, and T. Shindo. Ap1000+: Architectural Support of a put/get Interface for Parallelizing Compilers. In *Proc. of ASPLOS IV*, pages 196–207. ACM, Oct 1994.

[8] S. Hinrichs, C. Kosak, D. O'Hallaron, T. Stricker, and R. Take. An Architecture for Optimal All-to-All Personalized Communication. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 310–319, Cape May, New Jersey, June 1994. A revised version is available as Tech. Report CMU-CS-94-140.

[9] C. Leiserson, A. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. St.Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Symposium on Parallel Algorithms and Architectures*, pages 272–285, San Diego, June 1992. ACM.

[10] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994. ftp.cs.sandia.gov /pub/sunmos/papers/published/ISUG94-1.ps.

[11] G. McRae, W. Goodin, and J. Seinfeld. Development of a Second-Generation Mathematical Model for Urban Air Pollution - Model Formulation. *Atmospheric Environment*, 16(4):679–696, 1982.

[12] R. Numrich, P. Springer, and J. Peterson. Measurement of Communication Rates on the Cray T3D Interprocessor Network. In *Proc. HPCN Europe '94, Vol. II*, pages 150–157, Munich, April 1994. Springer Verlag. Lecture Notes in Computer Science, Vol. 797.

[13] W. Oed. The Cray Research Massively Parallel Processor System Cray T3D, 1993. Available from via ftp from cray.com.

[14] E. J. Schwabe, G. E. Blelloch, A. Feldmann, O. Ghattas, J. R. Gilbert, G. L. Miller, D. R. O'Hallaron, J. R. Shewchuk, and S. Teng. A Separator-Based Framework for Automated Partitioning and Mapping of Parallel Algorithms for Numerical Solution of PDEs. In *Proceedings of the 1992 DAGS/PC Symposium*, pages 48–62, June 1992. Revised version accepted for Comm. ACM.

[15] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, 1994.

[16] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. The Performance Impact of Fast Synchronization in Parallel Computers To appear in Proceedings of International Conference of Supercomputing, Barcelona, Spain, July 1995.

[17] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19th Intl. Conf. on Computer Architecture*, pages 256–266, May 1992.

Acknowledgements

We appreciate discussions with J. Brandenburg of Intel and S. Wheat of Sandia. J. Kyle and B. Numrich of Cray Research, G. Blelloch, G. Gibson, and S. Hinrichs of Carnegie Mellon provided comments on earlier drafts of the paper. S. Hinrichs, M. Hemy, and P. Dinda of Carnegie Mellon helped us understand the Paragon and contributed measurements of Paragon performance. D. O'Hallaron and the rest of the Quake Project consulted on the application codes. The staff of the Pittsburgh Supercomputing Center dealed graciously with our many requests.