The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors

Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

Integrating support for block data transfer has become an important emphasis in recent cache-coherent shared address space multiprocessors. This paper examines the potential performance benefits of adding this support. A set of ambitious hardware mechanisms is used to study performance gains in five important scientific computations that appear to be good candidates for using block transfer. Our conclusion is that the benefits of block transfer are not substantial for hardware cache-coherent multiprocessors. The main reasons for this are (i) the relatively modest fraction of time applications spend in communication amenable to block transfer, (ii) the difficulty of finding enough independent computation to overlap with the communication latency that remains after block transfer, and (iii) long cache lines often capture many of the benefits of block transfer in efficient cache-coherent machines. In the cases where block transfer improves performance, prefetching can often provide comparable, if not superior, performance benefits. We also examine the impact of varying important communication parameters and processor speed on the effectiveness of block transfer, and comment on useful features that a block transfer facility should support for real applications.

1 Introduction

A shared address space with coherent caches provides an effective communication abstraction for multiprocessors. The shared address space greatly simplifies the parallel programming task—particularly for irregular, dynamically changing communication patterns—by making communication implicit rather than requiring explicit communication in the user program. Furthermore, automatic coherent caching of shared data reduces the implicit communication often to levels that yield good performance.

One perceived shortcoming of existing cache-coherent architectures is that they rely on implicit communication through loads and stores as the *only* means of communication. While this works well for fine-grained data sharing patterns, applications often require the movement of a large amount of data from one processing node to another. In this case, moving the data in one large message in a pipelined fashion is likely to be more efficient than moving the data one cache line at a time through processor loads and stores. In addition to *fast pipelined data transfer*, block transfer offers other advantages, including *replication* of communicated data in the local memory of the receiver, and the ability to *overlap* the transfer with concurrent computation. Since block data transfer and a cache-coherent shared address space are clearly not mutually exclusive, and since the core mechanisms for moving data with low latency and high bandwidth are very similar for cache coherence and block transfer, architects have begun designing machines with hardware support for both a cache-coherent shared address space and block transfer of data [1, 9, 12, 14].

Structuring communication in large blocks is crucial on current message-passing machines and workstation networks, since the overheads and latencies of sending messages are very high. However, it is not clear what the role of coarse-grained messages is on tightly coupled, hardware cache-coherent shared address space architectures. There are a number of issues to investigate, such as: (i) the performance advantages of selectively using block transfer over relying completely on a standard load-store shared address space model, (ii) how these advantages vary with processor, memory system and network performance, (iii) how block transfer compares with other latency hiding techniques such as prefetching, (iv) the desirable capabilities of a block transfer mechanism, (v) whether algorithms must change substantially to effectively use block transfer, and (vi) the implications for the programming model in such an integrated architecture. In this paper, we focus primarily on the performance advantages of using block transfer, how these advantages change with varying architectural parameters, and how they compare against prefetching, by studying important scientific applications that appear likely to be aided by block transfer. We address block transfer features and algorithmic changes where they are relevant in the applications we consider. We do not address the programming model question, but use an explicit *memory copy* primitive to implement block transfer, enabling us to enhance performance without the overheads or restrictions of a specific high-level software system.

Section 2 discusses different ways to structure coarse-grained communication in a cache-coherent multiprocessor with a block transfer facility, and the advantages that can be obtained by using block transfer. Section 3 describes the architecture and simulation environment that we use in our experiments, as well as the values we use for various architectural parameters in our base machine. Our choice of applications and our experimental methodology are discussed in Section 4. Section 5 presents the results for different applications on the base machine. In Section 6 we examine the impact of varying communication parameters and processor speed on the relative effectiveness of block transfer. Section 7 discusses the performance of prefetching as an alternative to block transfer. Finally, Section 8 summarizes our main conclusions.

2 Communication Alternatives

Given a block data transfer facility in a cache-coherent shared address space machine, a programmer is presented with two mechanisms for communicating a large chunk of data: (i) via processor loads and stores (the load-store model), and (ii) via block transfer.

Coarse-grained communication through block transfer provides three major advantages over communication based solely on loads and stores:

- *Fast pipelined transfer* of large amounts of data. This is achieved since the overhead of message request, initiation, and management is incurred only once per block transfer rather than once per cache line transmitted. In addition, the datapath between main memory and the network is shorter through the block transfer facility than through the main processor.
- *Overlap of communication with computation* or with other communication. By explicitly placing block transfer requests in application code, block transfers can be scheduled to obtain maximum overlap with other communication or computation.
- *Replication of communicated data in local main memory.* Since our block transfers are implemented with memory copy primitives, data that are transferred are copied into distinct memory addresses in the local main memory of the receiving node. If these data are reused, cache misses to them can be satisfied from local memory rather than by repeated remote communication. This advantage is similar to that obtained in Cache-Only Memory Architectures (COMAs).

The goal of our block transfer versions is to reduce communication costs through fast data transfer and replication, and to hide the remaining costs as far as possible by overlapping communication with computation, and we analyze the extent to which we are able to accomplish these goals. Block transfer has other advantages, such as the ability to naturally combine synchronization and data transfer in a single message, but these are less important for our applications and we do not consider them.

Two points bear emphasis. First, *we are not restricted to using only one communication mechanism in a given application*, but are free to use whichever method is most appropriate for a particular communication. In fact, we rely on the implicit load-store model for all but block communication, thus retaining the major advantages of a cache-coherent shared address space. Second, given that we have a shared address space, the most appropriate way to perform block transfer is not through a *send* operation to a processor that requires a matching *receive*, but by performing a *block memory copy* from source addresses directly into destination addresses in the application's data structures. Support for this type of copying mechanism is provided in the Cray T3D [12], and is planned for other machines currently being developed [1, 9, 14].

Within the load-store or block transfer models, communication can be distinguished by whether it is initiated by the consuming processor (*receiver-initiated*) or by the producing processor (*sender-initiated*). In the load-store model, these types of communication occur through read and write operations to remote data, respectively. While sender-initiated load-store communication has performance advantages under certain conditions, receiver-initiated communication is usually more natural to the load-store programming paradigm and we focus on it here (see Section 5.5)[1]. The performance advantages of sender-initiated load-store communication depend intimately on details of the architecture and application and are discussed in [17].

In our block transfer versions, we use the type of communication that is most natural to the application, which is sender-initiated for all our applications except for Cholesky, where receiver-initiated block transfer has some important advantages (see Section 5.3).

3 Architecture and Simulator

Every node in the cache-coherent multiprocessor we simulate contains a processor with a single-level cache, a node controller, a network interface, and an equal fraction of the main memory (See Figure 1). The node controller contains a *programmable* processor which processes local memory accesses, standard directory-based cache-coherence protocol transactions and block transfers. The network interface connects the node to a 2-dimensional, bidirectional mesh network. The architecture is based closely on the Stanford FLASH multiprocessor that is currently being developed [9].

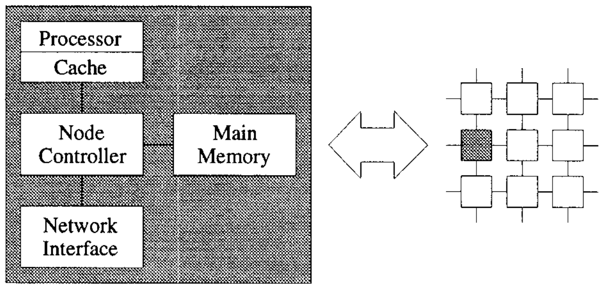


Figure 1: Multiprocessor Node Architecture.

We assume a set of ambitious hardware mechanisms for performing block transfer. The node controller is assumed to be quite sophisticated and allows different strides to be specified at the source and destination. This is feasible with a programmable node controller, since all the functionality does not have to be provided in hardware. Block transfers are handled entirely by the node controllers, so that both the source and destination main processors may continue to compute while the transfers are in progress, although they may contend for resources with block transfers. To perform a block transfer, the initiating processor describes the transfer to the node controller in its node. Data are then communicated by the node controller in a pipelined fashion at the granularity of cache lines [10]. The pipeline stage time for block transfer is the greater of the time for the node controller to retrieve a line from memory and the time to push a line and its associated header into the network. Synchronization for block transfer completion is performed by checking the status of a flag associated with the transfer. No restrictions are placed on the size or number of transfers that a processor can be involved in at a given time, and multiple transfers from a source are interleaved at a cache line granularity.

[1]We have experimented with sender-initiated communication, and it has advantages when (i) write buffers merge consecutive writes to the same remote line, (ii) writes occur far in advance of when the remote processor needs to access the written data, and (iii) cache sizes are relatively small so that when the remote processor accesses the data, it has been replaced from the writing processor's cache and written back to the local memory of the remote processor.

Parameter	Value
Main Processor Clock Rate	200 MHz
Peak Main Processor Instruction Issue Rate	1/cycle
Cache to Memory Bus Bandwidth	800 MB/sec
Queueing Delay into and out of Directory Controller	16 cycles
Memory Block State Lookup Time	30 cycles
Memory Block Remote Request Message Construction Time	20 cycles
Overhead for Cached Blocks at Sender and Receiver	10 cycles
Memory Block Access Time (independent of block size)	50 cycles
Network Width (each direction)	16 bits
Network Topology	2D Mesh
Network Clock Rate	200 MHz
Network Cache Line Header Size	32 bytes
Message Latency Between Adjacent Nodes	10 cycles
Block Transfer Message Startup Time	200 cycles

Table 1: Important Simulation Parameters and Their Values in the Base Architecture.

Application/Kernel	Representative of
Radix $\sqrt{n}$ Fast Fourier Transform	Convolution/Transform Methods
Blocked Dense LU Factorization	Dense Linear Algebra
Blocked Sparse Cholesky Factorization	Sparse Linear Algebra
Ocean Simulation with Multigrid Solver	Regular Grid Iterative Methods
Radix Sort	Sorting

Table 2: The Applications.

Four important issues arise when implementing block transfer on top of a cache-coherence protocol. First, block transfer data may be cached by a processor, requiring additional time to retrieve and/or invalidate the cached data. As in other proposed block transfer protocols, we assume that data can only be sent from the node on which its storage is allocated, and we ensure only that the sender and receiver's caches stay coherent with their respective local memories [7]. These assumptions suffice for all our applications. More ambitious protocols that maintain global coherence on block transfer data would add implementation complexity, and we comment on their importance in Section 5.2. The second issue is deciding how the node controller should prioritize load-store and block transfer transactions. Because block transfers may be large, priority is given to transactions related to load-store accesses. This allows for overlap of block transfer communication with main processor computation, which may generate load-store and other transactions. Third, a block transfer might include only a portion of a cache line rather than the full line. The destination node controller must be able to *merge* partially transmitted cache lines with the valid portion which may be cached or in memory. Fourth, block transfer can be performed into the receiver's cache or into its main memory. We perform block transfer into main memory for two reasons: (i) when communication is overlapped with computation, transferring into the cache might replace data that the processor is using, and (ii) replication in main memory is useful if the data needs to be reused later and does not stay in the cache. We shall explore transfers into a large second level cache in the future.

The architecture is simulated using the Tango-Lite event-driven reference generator [11]. Our detailed, variable-latency memory system simulator models contention at the node controller and memory system, but not in the network itself. An invalidation-based cache-coherence protocol similar to the one used in the Stanford DASH multiprocessor [13] is simulated. Processors are forced to block on read misses, but infinite write buffering hardware is included to eliminate processor stalls on write misses. To reduce miss latencies, speculative memory reads are performed at the home node at the same time that the state of a line is checked. The processor caches are fully associative to avoid artifacts due to cache conflicts, and the instruction cache hit rate is assumed to be 100%.

The important machine parameters we consider, as well as their values in the base architecture, are shown in Table 1. With 64-byte cache lines and no contention, the latency for a local read miss is about 80 processor cycles, the latency for a remote read miss is about 240 processor cycles, and the node-to-network interface has a peak bandwidth of about 380 MB/sec for block transfer.

4 The Applications and Methodology

4.1 Choice of Applications

Rather than attempt to compute an average effectiveness of block transfer over a broad range of computations, we carefully selected applications and kernels for this study based on two criteria: (i) they represent important classes of computations in scientific parallel computing, and (ii) they cover a range of scientific applications considered most likely to benefit from block transfer. Note that the four kernels we study are not *complete* applications, and that the rest of an application that uses them is not likely to benefit as much from block transfer as the kernels do. Thus, the performance gains we obtain through block transfer in these kernels will likely have a *smaller* impact on the overall performance of a complete application.

The applications and kernels we study are listed in Table 2. Our ocean simulation (which is a complete application) is a scalable version of the Ocean application in the SPLASH suite [16], and uses a different solver and partitioning scheme. The blocked Cholesky factorization is also a more scalable alternative to the panel Cholesky kernel in SPLASH.

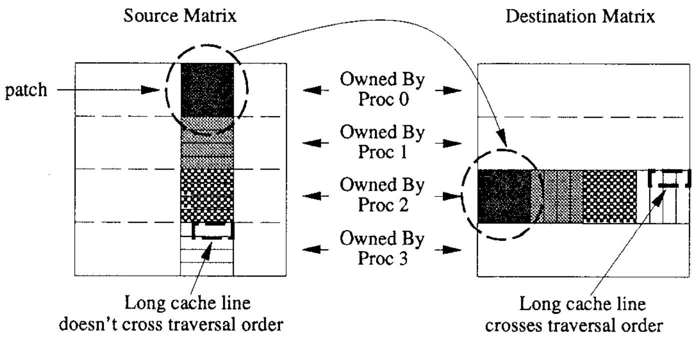
4.2 Experimental Methodology

For each application, we examine the performance gains obtained by block transfer over the load-store model. In many ap-

The Six–Step FFT Algorithm

```
1. Transpose data matrix
2. Perform 1–D FFT on each row
   of data matrix
3. Apply roots of unity to data
   matrix
      >>> Barrier <<<
4. Transpose data matrix
5. Perform 1–D FFT on each row
   of data matrix
      >>> Barrier <<<
6. Transpose data matrix
      >>> Barrier <<<
```

(a) Algorithm Steps



(b) FFT Transpose Phase

Figure 2: FFT Algorithm and Transpose Phase.

plications, there are several alternatives for incorporating block transfer, often at successive levels of implementation complexity and performance gain. While we have experimented with many intermediate block transfer versions, we do not discuss them here for reasons of space. A more complete discussion can be found in [17]. We start with highly optimized load-store versions of the applications, discuss the most effective ways to incorporate block transfer, and examine the performance benefits, trying to isolate their sources as much as possible. We also examine how the effectiveness of block transfer changes with the number of processors used. We choose cache sizes based on the working sets of the applications, so that the cache size is large enough to accommodate the important working set if this is likely to be the case in practice. In addition, we vary the cache line size from 32 to 128 bytes for all our applications.

In the individual application sections that follow, we use the base set of machine parameters in Table 1. These represent parameter values which might be found on a machine that can be built today. Then, in Section 6, we examine the effects of varying certain key parameters to reflect possible technology trends or systems which are more or less tightly coupled. Finally, we examine software-controlled prefetching as an alternative to block transfer in Section 7.

5 Base Architecture Results

For each application and kernel, we briefly describe the algorithm and the load-store (LS) and block transfer (BT) implementations for which we present results.

5.1 The Fast Fourier Transform

Algorithm: The FFT we use is a complex 1-D version of the radix-$\sqrt{n}$ *six-step* FFT algorithm described in [2], which is optimized to minimize interprocessor communication. The data set for the FFT consists of the n complex data points to be transformed, and another n complex data points referred to as the *roots of unity*. Both sets of data are organized as $\sqrt{n}$ x $\sqrt{n}$ matrices, and the matrices are partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. The six steps of the algorithm are shown in Figure 2(a). Communication occurs in the three matrix transpose phases (shown pictorially in Figure 2(b)), which require all-to-all interprocessor communication. Every processor transposes a *patch* (contiguous submatrix) of size $\frac{\sqrt{n}}{p}$ x $\frac{\sqrt{n}}{p}$ from every other processor, and transposes one patch locally.

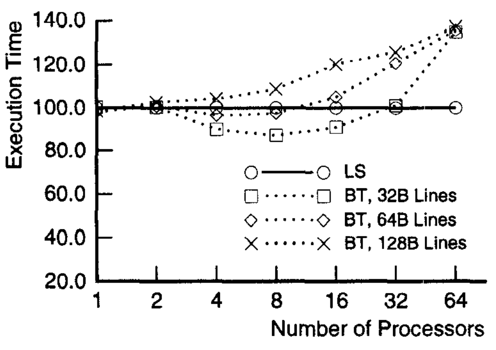
Load-store and block transfer versions: Instead of performing a matrix transpose by reading an entire column at a time of the source matrix and writing it into a row at a time in the destination matrix, the LS version utilizes a blocked transpose algorithm to exploit the spatial locality afforded by long cache lines [17]. Patches are communicated in a staggered fashion (processor i first transposes a patch from processor $i+1$, then one from processor $i+2$, etc.) in order to avoid hot-spotting.

A simple BT implementation of the transpose phase might transfer one subrow of a patch at a time and put it in the proper subcolumn at the destination, utilizing different strides at the source and destination[2]. Figure 3(a) shows the performance of this version. Unless otherwise specified, all graphs show performance of a BT version relative to the LS version. In each graph, the y-axis value for each curve is the BT version's execution time for a given number of processors and a given line size, normalized to the execution time of the LS version for the same number of processors and the same line size. Thus, every point shows at a glance how much better or worse the BT version is relative to the LS version *for the same set of parameters.*

Results: While the BT version in Figure 3(a) does well for small cache lines and small numbers of processors, it performs much worse as the line size and number of processors increase. There are three reasons for this. First, the transposes using BT are not blocked for cache line reuse as in the LS version. For larger line sizes, more patch elements are contained on each line, and the amount of reuse obtained in the LS version is greater. Second, the destination node receives many partially valid lines, since a cache line may straddle consecutive subcolumns, decreasing performance as discussed earlier. Finally, increasing the number of processors for the same problem size results in more patches. This means smaller patch subcolumns and hence smaller transfers, resulting in a greater impact of block transfer overhead.

Enhancing block transfer: These three problems can be overcome by taking advantage of the flexibility of the programmable node controller. In the LS version, the major source of performance improvement is blocking of the patch transposes. To provide the same capability in the BT version, a special software *block transfer handler* can be written that blocks

[2]Another possibility is to transfer an entire patch, and have the receiver transpose the received patch before using it. Transferring a patch in a single message requires more than a stride capability: It also requires the ability to specify the number of elements at the stride points. In addition, this method requires the main processor to do the transpose, and to use additional buffer space. Finally, the method of blocking in the transfer engine discussed next is the best method anyway.



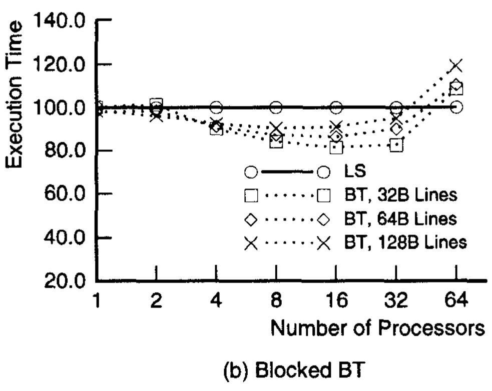
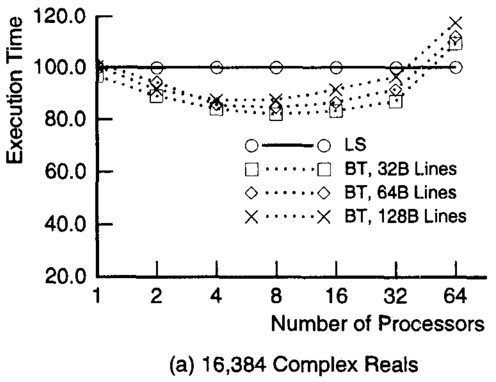


Figure 3: FFT: Unblocked BT and Blocked BT versus Blocked LS (16,384 Complex Reals, 8KB Cache).



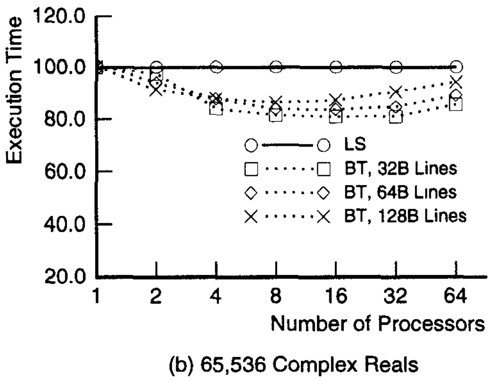


Figure 4: FFT: Subpatching to Allow Overlap of Communication and Computation (8KB Cache).

the transpose in the node controller rather than in the main processor. It is possible to provide this ambitious capability with a programmable node controller, and the results for this version are shown in Figure 3(b). With this support, the BT version exhibits improved performance for all cache line sizes, although the magnitude of improvement is greatest for the largest line sizes where the amount of cache line reuse in the blocking handler is greatest.

Overlapping communication with computation: The BT version with a blocked transpose obtains the benefit of fast data transfer, but not overlap of communication with computation at a coarse level. Through a technique we call *subpatching* [17], overlap can be obtained by combining steps in the six-step algorithm. The patches to be transposed are first broken into smaller *subpatches*. After the row-wise FFTs are computed on the rows which span a set of subpatches, these subpatches are block transferred by the node controller while the main processor performs row-wise FFTs on the next set of subpatches.

Subpatching Results: Figure 4 illustrates the performance of the subpatching method for two different problem sizes. There is a *sweet spot* in the number of processors at which the effectiveness of block transfer is greatest for a given problem size. Increasing the number of processors beyond this point increases the communication to computation ratio, but reduces the individual block transfer size so that block transfer overhead begins to dominate and performance in reduced. Decreasing the number of processors reduces the communication to computation ratio and hence the importance of block transfer.

As problem size increases, the sweet spot shifts towards larger numbers of processors. This is because patch sizes increase and block transfer overhead is thus amortized over more elements. However, beyond a point, the *magnitude* of performance improvement does not increase. Once block transfers become large enough that their startup overhead is effectively amortized, the improvement in communication speed is limited by the ratio of the remote read miss time in the load-store case to the pipeline stage time in the block transfer case.

Finally, as in almost all our applications, the relative advantage of block transfer is greater with smaller cache lines. The FFT utilizes the increased spatial locality afforded by long cache lines very effectively, reducing the processor cache miss rate (and hence the communication cost) in the LS case. In addition, as partitions become smaller, longer cache lines can lead to partially transmitted cache lines, which hurt block transfer substantially.

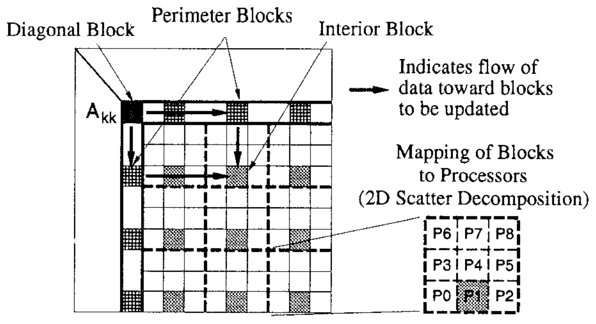
Summary: Although the FFT appears to be a prime candidate for block transfer, even an ambitious block transfer mechanism did not gain very much over a load-store implementation, particularly with the long cache lines that modern machines are moving towards. Of the three primary advantages of block transfer, fast data transfer was the most useful, and a small additional increase in performance was obtained through overlap of communication with computation via our subpatching technique. Replication of communicated data was not an issue in this application.

5.2 Dense LU Factorization

Algorithm: *LU* factorization of a dense matrix is representative of many dense linear algebra computations, and can be performed efficiently if the dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks, ($n = NB$). Unlike the FFT application, blocking is performed in LU to exploit *temporal* locality on individual submatrix elements. The pseudo-code in Figure 5(a), expressed in terms of blocks, shows the most im-

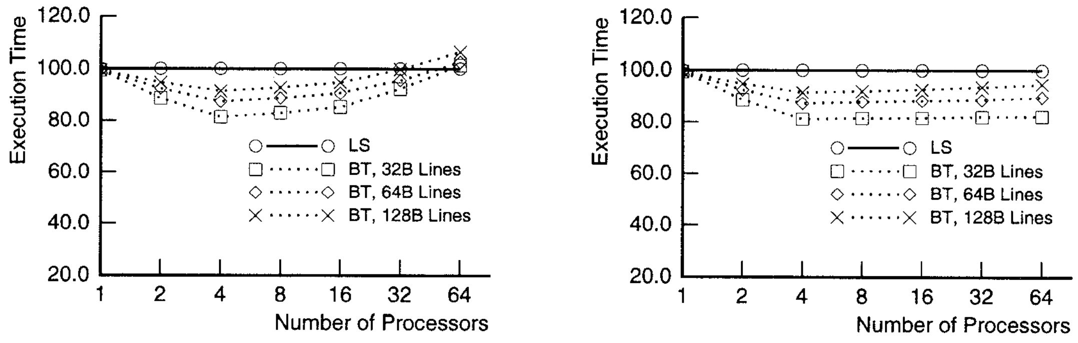
```
Blocked Dense LU Factorization
1. For k=0 to N-1 do
2.   Factor diagonal block $A_{kk}$
       >>> Barrier <<<
3.   Update all perimeter blocks in
     column k and row k using $A_{kk}$
       >>> Barrier <<<
4.   For j=k+1 to N-1 do
5.     For i=k+1 to N-1 do
6.       $A_{ij} = A_{ij} - A_{ik} * A_{kj}$
     /* Update interior blocks using
        corresponding perimeter blocks */
```

(a) Algorithm Steps


Perimeter Blocks
Diagonal Block
Interior Block
A_{kk}
Indicates flow of data toward blocks to be updated
Mapping of Blocks to Processors (2D Scatter Decomposition)
P6 P7 P8
P3 P4 P5
P0 P1 P2

(b) Pictorial Representation

Figure 5: Blocked Dense LU Factorization.



(a) Base BT

(b) BT with Broadcast

Figure 6: LU: Comparison of BT Methods versus LS No Replication (256x256 Matrix, B=16, 8KB Cache).

portant steps in the computation. The dominant computation is Step 6, which involves a dense matrix multiplication of two blocks.

The parallel computation corresponding to a single k iteration in the pseudo-code is shown symbolically in Figure 5(b). Two details are important for reducing interprocessor communication and thus obtaining high performance. First, the blocks of the matrix are assigned to processors using a 2-D scatter decomposition, an example of which is shown in Figure 5(b). Second, the matrix multiplication in Step 6 is performed by the processor that owns the destination block $A_{i,j}$. The block size B is chosen to be large enough to keep the cache miss rate low, and small enough to reduce the time spent in the less parallel parts of the computation (Steps 2 and 3) and to maintain good load balance in Step 6. In practice, relatively small block sizes ($B = 8$ or $B = 16$) strike a good balance.

Load-store and block transfer versions: In a shared address space, the natural data structure for the 2-D matrix being factored is a 2-D array. Since blocks are allocated in a scatter decomposition, and since a block is not contiguous in the address space in a 2-D array, it is difficult to allocate blocks in the local memories of the processors that own them. For the same reasons, false sharing problems can occur with long cache lines. These problems exist for the load-store model, but cause even greater complications for block transfer since memory allocation problems and false sharing imply that global coherence is required for correctness (see Section 3). One solution to ensure that blocks assigned to a processor are allocated locally and contiguously is to use a 4-D array, in which the first two dimensions specify the block number in the 2-D grid of blocks, and the next two specify an element in that block. This data structure allocates block elements in contiguous memory locations, and thus eliminates false sharing and the need for global coherence. For these reasons, it is used in all our versions of LU.

Communication occurs in LU when a diagonal block is used by all processors which require it to update the perimeter blocks they own, and when perimeter blocks are used by all processors that require them to update their interior blocks (see Figure 5). The block transfer mechanism inherently replicates communicated blocks in the local memory of the processors that need them. In our LS version, however, we found that the benefits of explicit replication in main memory are small and that the overheads often outweigh them [17]. Therefore, our LS version has no explicit replication.

In the BT version, the processor that updates the diagonal block sends a copy of it to all processors that own perimeter blocks. When a processor updates a perimeter block, it sends a copy to all processors that own interior blocks which need it, and proceeds to update the next perimeter block it owns (thus achieving some overlap between communication and computation). Having sent all its perimeter blocks, it waits until all incoming block transfers have completed and then updates its interior blocks.

Results: Figure 6(a) shows that there is a sweet spot in LU for block transfer as well. In the FFT, the sweet spot is due to processors being involved in increasing numbers of simultaneous transfers, each of decreasing size, as the number of processors p increases. However, in LU messages stay the same size. The sweet spot is due to processors being involved in increasing numbers of simultaneous transfers (growing as $\sqrt{p}$ rather than p as in the FFT), and the fact that with larger numbers of processors load imbalance becomes the dominant performance bottleneck. The effectiveness of block transfer also diminishes with increasing cache line size as in the FFT.

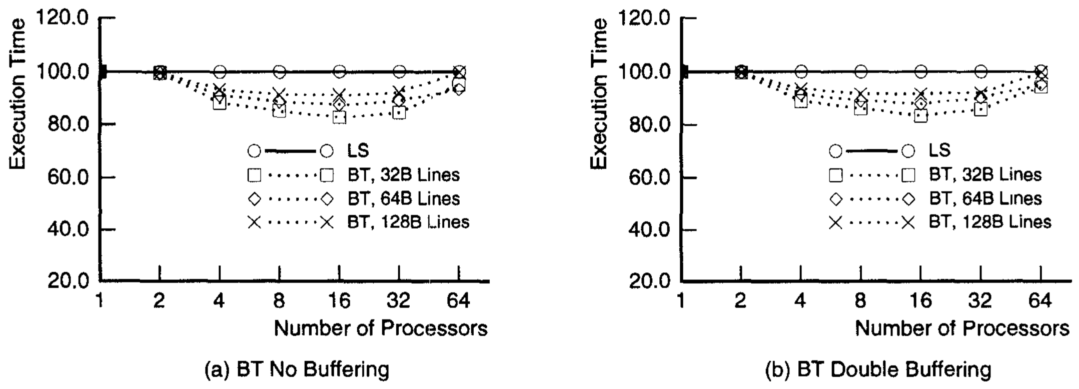


Figure 7: Cholesky: Comparison of BT Buffering Strategies versus LS No Buffering (BCSTTK15, B=32, 16KB Cache).

Enhancing block transfer: With a regular scatter decomposition, every processor sends blocks only to processors in the same row or column of the processor mesh. This suggests that support for *broadcasting* block transfers to all processors in the same row, the same column, and the same row and column (for diagonal blocks) may be beneficial. Broadcast reduces the number of block transfers a processor is simultaneously involved in by a factor of $\sqrt{p}$, thus reducing overhead and contention at the node controller of the broadcaster. Figure 6(b) shows that broadcast improves performance substantially for larger numbers of processors.

Summary: Although block transfer provides fast data transfer, overlap of communication with computation, and replication in main memory for LU factorization, it still does not have a very large effect on performance, especially when long cache lines are used. There are two reasons for this. First, the use of blocking yields a low cache miss rate, resulting in a low communication to computation ratio to begin with. Second, it is difficult to find a good sweet spot where the communication to computation ratio is high enough for block transfer to have substantial gains, yet the number of blocks per processor is high enough that load imbalances don't become the dominant effect limiting performance.

5.3 Blocked Sparse Cholesky Factorization

Algorithm: Blocked sparse Cholesky factorization is similar in structure and partitioning to blocked dense LU factorization, but has two major differences: (i) it operates on sparse matrices, which makes the communication to computation ratio much larger, and (ii) to achieve good load balance and performance, it is not globally synchronized between steps like LU. This second difference implies that the amount of replication needed at any given time at a receiving node is not known a priori or not well-bounded in sender-initiated block transfer, since one processor may get multiple steps ahead of another. Also, the order in which a processor sends blocks may not be the order in which a receiver needs to use them. *Sender-initiated communication is therefore not appropriate*, and we use receiver-initiated block transfers.

Load-store and block transfer versions: We use the benchmark matrix BCSTTK15 from the Boeing-Harwell suite for our experiments, since it is small enough to simulate but large enough to be realistic. We use a 16KB cache, which holds the data required for a block-block update with our block size 32-by-32. As in LU, we found that main memory replication was not useful in the LS versions, and use a version with no explicit replication as the base [17].

In receiver-initiated block transfer, the receiver sends request messages to the node controller of the block's owner, which then transfers the block to it. In developing the BT version, we examined three replication strategies: (i) the receiver has only one block-sized receive buffer, so that a received block is immediately discarded after it is used (and may need to be requested again later), (ii) the receiver has two block-sized receive buffers, one of which can be used for computation while data is transferred concurrently into the other (a *double-buffering* approach that achieves overlap of communication and computation through block *prefetching* [17]), and (iii) the receiver utilizes user-level memory management (without system calls) to locally replicate all remote blocks that it requires (a *full replication* approach). The first version obtains fast transfer of data, the second obtains fast data transfer and overlap of communication with computation, and the third obtains fast data transfer, overlap, and replication of communicated data.

Results: Dynamic memory management for the full replication BT version hurts performance significantly as it did in the LS version, and the performance of this version is not shown. Results for the other versions are shown in Figure 7. While fast data transfer helps reduce communication costs, overlap through double-buffering provides no noticeable improvement for two reasons. First, a processor finds blocks to prefetch only 30-40% of the time it looks for them. Second, the prefetched block often has not arrived by the time the processor requires it. Overall, blocked sparse Cholesky factorization also does not benefit much from block transfer, particularly with long cache lines.

5.4 Ocean Simulation

Application description: The ocean simulation studies large-scale ocean movements based on eddy and boundary currents, and is an enhanced version of the Ocean application in the SPLASH application suite [16]. The major differences between this version and the SPLASH version are: (i) it is written in C rather than FORTRAN, (ii) it partitions the grids into square-like subgrids rather than groups of columns to improve the communication to computation ratio, (iii) as in LU, it uses dynamically allocated 4-D arrays designed to allow appropriate data distribution and reduce false sharing, and (iv) it uses a red-black Gauss-Seidel multigrid technique based on that presented in [4], whereas the SPLASH version uses a relaxed Gauss-Seidel SOR solver.

Results: The bulk of the application's execution time is spent in the multigrid solver. The solver performs nearest-neighbor sweeps on a hierarchy of grids rather than on a single grid as in SOR. Higher grids in the hierarchy have less points in them. Since communication occurs at partition boundaries at all levels of the hierarchy, the communication to computation ratio becomes larger at higher levels. While this should increase the advantage of block transfer at higher levels, these levels have smaller partition borders and hence smaller block transfer messages, resulting in an increased effect of block transfer overhead. At lower levels, the communication to computation ratio is typically small. It is therefore difficult to find a point where block transfer is very useful. Also, other parts of the application than the multigrid solver do not have nearly as much communication. Results for this application show negligible BT performance benefits and may be found in [17]. Regular-grid nearest-neighbor problems such as Ocean represent a dominant class of applications with communication that is quite natural to block transfer, but which do not benefit much from block transfer in a tightly-coupled multiprocessor.

5.5 Radix Sort

Algorithm: The integer radix sort kernel also has a wide range of applications, including database management systems and aerospace applications (it is one of the NAS parallel benchmarks [6]). It is based on the method described in [8], and requires the movement of bulk data (the keys being sorted) from one processor to another during each phase of the computation. Each processor is assigned an equal fraction of the n keys to be sorted. The algorithm is iterative, performing one iteration for each radix r digit in the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram of values. Next, the local histograms are accumulated into a global histogram, which is used to permute the keys for the next iteration. Keys are communicated in the permutation step, and the permutation is inherently a sender-determined one, so *receiver-initiated communication does not make sense* in either the load-store or block transfer paradigms.

Load-store and block transfer versions: The performance of radix sort is strongly dependent on the choice of radix. A larger radix implies less digits and hence fewer iterations through the algorithm. However, a larger radix also implies larger histograms and hence larger cache requirements. In the BT version, there is an additional constraint. Processors must perform at least one block transfer for every radix value in each iteration, so a small radix implies few large transfers, while a large radix implies many small transfers and hence greater overhead. The BT version also requires an additional step in each iteration and an associated data structure to gather together the keys that must be communicated to the same processor. We empirically determine the optimal radix separately for the LS and BT versions.

Results: Detailed results for radix sort may be found in [17]. The BT version tends to have nearly the same optimal radix as the LS version in our experiments, but does not perform as well for three reasons. The first is the problem associated with small messages mentioned earlier. Second, communication is done through writes in the LS version, which allows a fair amount of the communication latency to be hidden even with realistic write buffers. The third reason is the overhead of the additional step required in each iteration of the BT version. Since every processor performs at least r block transfers of average size $\frac{n}{rp}$ in each iteration, a problem configuration that gives block transfer significant advantages would require both a *very* large number of keys *per processor per radix value* and a similar optimal radix for the BT and LS schemes.

6 Effect of Architectural Variations

For our base architectural parameters, we found that while block data transfer helped improve performance in some situations, the improvement was not large especially when long cache lines were used. In fact, the effectiveness of block transfer depends on architectural parameters such as network bandwidth, communication overhead, and processor speed. In this section, we examine the effects of varying these three parameters to reflect technology trends or other types of systems that are more or less tightly coupled.

6.1 Varying Communication Bandwidth

We vary network bandwidth by changing the clock cycle time of the network. We experiment with increasing the network speed by a factor of four, which changes the ratio of one-hop remote miss latency to local miss latency from 3-to-1 in our base machine to 2-to-1. We also decrease network speed by a factor of four, which makes the ratio 4-to-1.

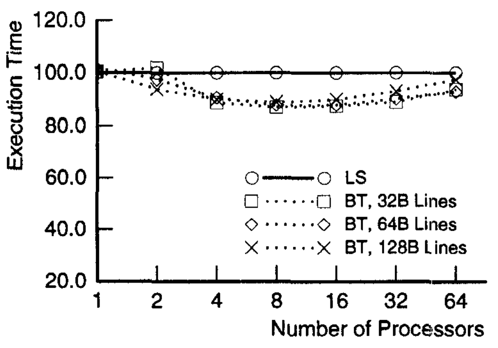
Results: Moderate changes in network speed clearly have little impact on the effectiveness of block transfer for applications with low communication to computation ratios, such as LU. However, they can have a stronger impact on applications with high communication to computation ratios or all-to-all communication, such as FFT. The effectiveness of block transfer is limited by the ratio $\frac{R}{P}$, where R is the remote read miss time in the load-store case and P is the block transfer pipeline stage time. The block transfer pipeline stage time is the maximum of the time for the node controller to access a block in local memory and the time to inject the block and its header into the network, while the remote read miss time is influenced by the sum of these two factors. Increasing the network bandwidth reduces network injection time and thus remote miss time. Beyond a point, it does not affect the block transfer pipeline stage time, which becomes limited by the memory block access time. Further increases in network bandwidth thus increase LS performance, but BT performance does not increase.

More interestingly, decreasing communication bandwidth (and hence coupling the machine more loosely) also has a detrimental effect on block transfer. Decreasing the bandwidth increases the remote read miss time, and beyond a point, it also increases the pipeline stage time, thus reducing the effectiveness of block transfer. Our base parameters are such that the magnitudes of these two factors are about equal. For the aforementioned reasons, large increases or decreases in network bandwidth reduce the effectiveness of block transfer. Compared to Figure 4(b), Figure 8(a) illustrates the decreased relative performance for FFT from increased network bandwidth. Results for decreased network bandwidth are not shown but are quantitatively similar to those shown in Figure 8(a).

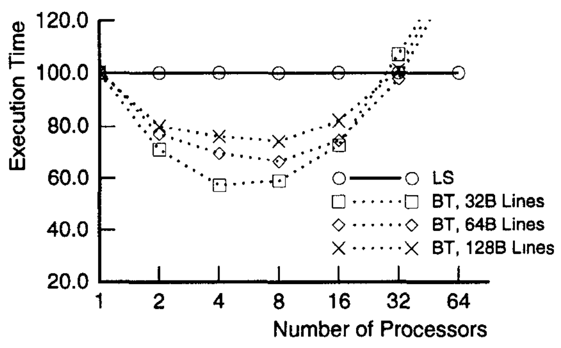
6.2 Increasing Communication Overhead

Another type of looser machine coupling is one in which the overhead of initiating communication with remote nodes is larger. To simulate this scenario, we increase the time required to initiate a remote request to another node from 20 cycles to 400 cycles, resulting in a one-hop remote to local miss ratio of 8 to 1. The block transfer startup time is increased proportionally from 200 cycles to 4000 cycles. The higher overheads move us towards systems in which communication is not as closely integrated into the processing nodes, such as current message-passing machines and networks of workstations.

Results: Increased communication overhead makes block transfer very helpful, particularly for codes with high communication to computation ratios or all-to-all communication. Compared to Figure 4(b), Figure 8(b) illustrates the improved relative BT performance for FFT resulting from increased communication overhead. The main reason is the fast pipelined transfer of data associated with block transfer. Every remote miss in the LS version incurs the increased overhead, whereas in the BT versions the increased startup time is amortized over the length of the entire block transfer. However, increased

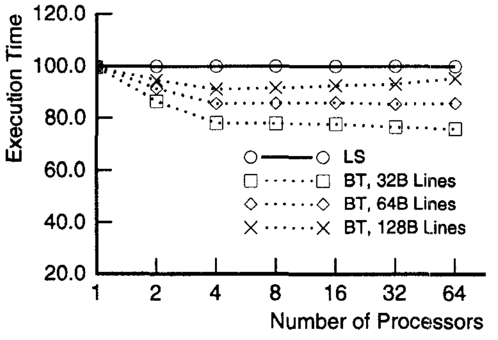


(a) Lower Communication Latency via Higher Network Bandwidth

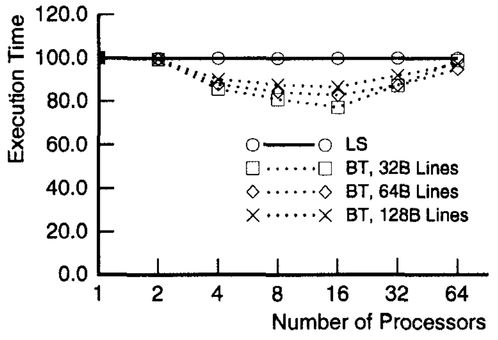


(b) Higher Communication Latency via Higher Communication Overhead

Figure 8: FFT: Varying Communication Latency (65,536 Complex Reals, 8KB Cache; compare with Figure 4(b)).



(a) LU (BT with Broadcast)



(b) Cholesky (BT with Double Buffering)

Figure 9: Impact of Faster Processor on LU and Cholesky (compare with Figures 6(b) and 7(b)).

overhead moves the sweet spot towards smaller numbers of processors. This is because larger numbers of processors make transfers smaller and hence make it more difficult to amortize the increased overhead. Finally, the BT curves for different line sizes in Figure 8 move further apart from each other as communication latency increases. This is because there are more cache misses for smaller line sizes, and each miss incurs the increased communication overhead in the LS case, making block transfer all the more important.

Conclusions: With long cache lines, we find that it takes large communication *overheads* to obtain substantial benefits from block transfer, overheads that are unrealistic for hardware cache-coherent multiprocessors. These trends, however, reconfirm the importance of block transfer in less tightly coupled systems that have much higher communication overhead. In a shared address space implemented in software on a typical message-passing system, the one-hop remote to local miss ratio might typically be in the range of 20-50 to 1, while on a network of workstations that ratio today would certainly exceed 100 to 1. For this reason, block transfer has enormous advantages in such systems [3, 5].

6.3 Increasing Processor Speed

We next examine the effect of having processor speeds increase much faster than memory and network speeds, a continuing technology trend. In particular, we show results for processors that have a peak performance twice that of our base processor.

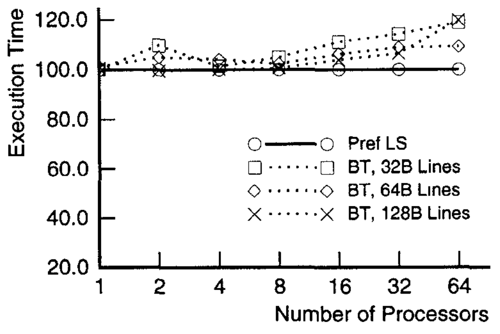
Results and conclusions: Increased processor speed has little impact on applications that have high communication to computation ratios, since performance is already dominated by communication time. A faster processor increases the relative BT performance in codes with a low communication to computation ratio (such as LU and Cholesky) since computation time is reduced and communication time becomes a larger percentage of overall execution time. Figure 9 illustrates the effects on LU and Cholesky. Despite the increased communication to computation ratio, LS performance is still good in these codes because blocking yields low miss rates.

7 Prefetching

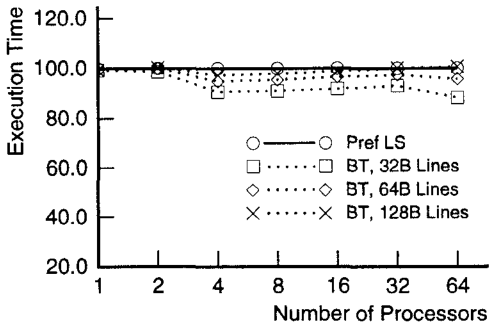
One alternative to block transfer in communication performance is prefetching into the processor cache. To determine its effectiveness, the base LS versions of codes that showed benefit from block transfer (FFT, LU, Cholesky) were augmented to perform the same communication with receiver-initiated prefetches. Note that we only prefetch references that require interprocessor communication, since we are interested in prefetching only as an alternative to block transfer for communication. We hand-code all prefetches, and assume no contention in the main processor caches between returning prefetch data and processor accesses.

The advantage of prefetching is that data are placed in the cache rather than in local memory, resulting in a substantially smaller access time. In the block transfer case, however, data are transferred into main memory and the first subsequent access to each line of the transfer incurs the local miss latency.

Results: Figure 10 shows the results of our prefetching experiments. In the transpose phase of FFT, data that is prefetched is used almost immediately after the prefetch completes, causing prefetching to outperform block transfer. In LU, however, the BT version outperforms the LS version with prefetching. The reason is that prefetching is initiated by the receiver, and hot-spotting results when processors that own perimeter blocks simultaneously issue prefetches to the processing node that owns



(a) FFT
65,536 Complex Reals, 8KB Cache



(b) LU (BT with Broadcast)
256x256 Matrix, B=16, 8KB Cache

Figure 10: Effectiveness of Prefetching versus Block Transfer for FFT and LU.

the diagonal block. The BT version avoids hot-spotting since communication of the diagonal block is done through a broadcast mechanism (see Section 5.2). A similar situation occurs with communication of the perimeter blocks. Prefetching provides minimal performance improvement in the LS version of Cholesky for similar reasons.

Conclusions: Prefetching can be more effective than block transfer when data that are prefetched do not cause hot-spotting and when they are used shortly after the prefetches complete. Even in cases where hot-spotting of prefetches occurs, prefetching still provides some performance benefit, and can help to close the gap in performance relative to block transfer.

8 Summary and Conclusions

We have studied the performance benefits of providing a block transfer facility within cache-coherent shared address space multiprocessors. We had set out to address three questions: (i) what are the performance benefits of block transfer, (ii) do algorithms need to change significantly to achieve good performance with block transfer, and (iii) what special features are useful to provide in a block transfer facility. We now summarize our answers to these questions in reverse order.

Block transfer features: We found flexibility of the block transfer facility to be useful. The ability to specify different source and destination strides, to write special block transfer handlers (such as the blocked transpose handler for FFT), and to broadcast transfers were all important. We did not need to maintain global coherence on block transfer data in our implementations, and found that false sharing of cache lines would have been the major reason for requiring global coherence in less optimized versions of the applications (see Section 5.2). The block transfer mechanism we have evaluated is quite general and is likely more ambitious (or higher performance) than might be available in real machines. Thus, our performance results for block transfer are likely to be optimistic for the class of tightly-coupled multiprocessors we have considered.

Algorithmic changes: The BT version of FFT benefitted significantly from algorithmic changes to exploit overlap of communication with computation. Sparse Cholesky factorization used double-buffering to exploit overlap but did not benefit much from it, and radix sort required an additional gather step which decreased performance. The other applications were conceptually straightforward extensions of the best load-store algorithms.

Performance results: Overall, block transfer did not help performance very much for our tightly coupled systems. There are several reasons for this. First, the fast data transfer advantage of block transfer is fundamentally limited by the ratio of the remote read miss time in load-store communication to the pipeline stage time in block transfer. Current memory system designs are moving towards larger cache line sizes. Many applications, particularly those that are amenable to block transfer, are able to exploit the spatial locality of longer cache line sizes well. Exploiting spatial locality effectively amortizes the remote read miss time in the LS case, thus reducing the effectiveness of block transfer. In addition, long cache lines can hurt block transfer by requiring partially transmitted lines to be merged at the receiver. The second reason is that the fraction of execution time spent in communication amenable to block transfer was often not very large in the kernels, and will likely be even smaller for a complete application which uses them. And when the problem size per processor was small enough to make this fraction large, the overheads of small block transfers or other factors such as load imbalance often began to dominate. Third, even though block transfer reduced communication costs, we were unable to overlap enough independent computation with the remaining communication. This problem becomes more severe as processor speeds increase relative to communication bandwidth.

Of the three major advantages of block transfer, fast data transfer provided us with most of our performance benefits. Overlap of communication with computation was not very useful for Ocean and radix sort, had a small effect for LU and Cholesky, and had a more substantial effect for FFT. Replication of transferred data in main memory was not relevant in three applications (Ocean, FFT, and radix sort) and its benefits did not outweigh its costs in others (LU and Cholesky)[3]. Contention at the node controller and memory between the block transfer and load-store accesses was in all cases not a significant problem compared to other sources of performance loss.

Scaling: Scaling arguments further diminish the potential effectiveness of block transfer. Under memory-constrained scaling, the communication to computation ratio often stays constant. In some cases (e.g. Ocean and LU), the message size and number of block transfers a processor is involved in stays fixed or grows slowly, so the performance advantages of block transfer persevere. In other cases that have all-to-all communication (FFT and radix sort), the message size is inversely proportional to the number of processors, so that a processor is involved in increasing numbers of block transfers of decreasing size, and the advantages of block transfer diminish even under memory-constrained scaling. The most realistic scaling model for most applications, however, is time-constrained scaling, in which the data set size does not grow as quickly as the number

[3]Replication in main memory can be useful for data structures that are not block transferred (particularly in the presence of conflict misses), and when data distribution is not done appropriately.

of processors [15]. The communication to computation ratio grows, but the message size becomes smaller, so that beyond a point the effectiveness of block transfer usually diminishes under time-constrained scaling.

Effect of architectural variations: Moderate changes in network bandwidth did not affect the benefits of block transfer very much. More substantial changes in network bandwidth have an impact on the advantages of block transfer in applications with all-to-all communication (FFT), and increased processor speed improves the advantages for applications with low communication to computation ratios (LU and Cholesky). The greatest advantages for block transfer, however, occur in machines with a high overhead to initiate communication. Thus, block transfer is expected to be useful in machines that provide only low performance implementations of shared memory, including message passing machines and networks of workstations.

Prefetching: We found that in the cases where block transfer increased performance, software-controlled prefetching often achieved similar, if not superior, performance. The primary advantage of prefetching is that data are brought into the cache rather than into main memory, and its primary disadvantage is that it can lead to hot-spotting. In particular, prefetching was better when the prefetch requests were distributed to the memory systems of all processors (FFT) than when they were concentrated towards one memory system (LU).

Conclusion: Despite the recent trend to incorporate block data transfer facilities and message passing in shared address space multiprocessors, our results show that block data transfer may not be very helpful in increasing the performance of well-written applications on efficient cache-coherent machines. Block transfer can be beneficial if the *overhead* of communicating with remote nodes is high, as in traditional message-passing machines and networks of workstations. Block transfer may also be useful for other purposes, such as implementing the message-passing programming model on a shared memory machine, as well as for special-purpose tasks such as operating system page migration or block memory copy.

Future work: In the future, we plan to investigate block transferring data into a large second-level cache as an alternative to transferring into local main memory. We also plan to study the relative performance of block transfer for low-associativity caches and versus load-store implementations with finite-depth write buffers. Finally, we shall look for other applications that might benefit from block transfer.

Acknowledgements: The authors wish to thank the Stanford FLASH multiprocessor design team for many helpful discussions. Steven Woo is supported by a Hughes Aircraft Company Doctoral Student Fellowship. J. P. Singh and John Hennessy are supported under ARPA contract number N00039-91-C-0138.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[2] David H. Bailey. FFTs in External or Hierarchical Memory. *Journal of Supercomputing*, 4(1):23–35, March 1990.

[3] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of COMPCON'93*, February 1993.

[4] Achi Brandt. Multi-Level Adaptive Solutions to Boundary-Value Problems. *Mathematics of Computation*, 31(138):333–390, April 1977.

[5] Sandhya Dwarkadas, Pete Keleher, Alan Cox, and Willy Zwaenepoel. An Evaluation of Software Distributed Shared Memory for Next-Generation Processors and Networks. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, May 1993.

[6] David H. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[7] David Kranz et al. Integrating Message-Passing and Shared-Memory: Early Experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

[8] Guy E. Blelloch et al. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[9] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[10] John Heinlein et al. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, October 1994.

[11] Stephen Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance.* PhD thesis, Stanford University, June 1993.

[12] Cray Research Inc. Cray T3D System Architecture and Overview. Revision 1.c. Technical report, Cray Research Inc., September 1993.

[13] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[14] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[15] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, 26(7):42–50, July 1993.

[16] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992. Also Stanford University Technical Report No. CSL-TR-92-526, June 1992.

[17] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors. Technical Report CSL-TR-93-593, Stanford University, December 1993.