Data Reshuffling in Support of Fast I/O for Distributed-Memory Machines

Claudson Bornstein

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Peter Steenkiste

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Achieving high-speed network I/O on distributed-memory systems is a hard problem because their architectures are, in general, ill-suited for communication processing. One of the problems is that messages are distributed over the private memories of the distributed-memory system. This can result in poor performance since communication includes a complex scatter/gather operation. This paper presents a strategy in which the task of creating large contiguous messages is performed on the distributed-memory system, thus minimizing the overhead on the network interface. The performance results for an implementation of this strategy for the iWarp system are presented.

1 Introduction

Supercomputer applications have to communicate over a high-speed network to, for example, display results on a framebuffer, read or write data from storage, or interact with other computer systems as part of a distributed computing application. In the last few years, networks based on the ANSI High-Performance Parallel Interface (HIPPI) protocol [1] have become very popular in supercomputer centers, and several other high-speed network standards are in various stages of development by standards bodies. These include ATM (Asynchronous Transfer Mode) [2] and Fibre Channel [3].

Meanwhile, distributed-memory computer systems [4], [5], [6], [7], [8] are becoming widely used. And that is happening because they are inherently scalable and provide relatively inexpensive computing cycles when compared with traditional uniprocessor or shared-memory multiprocessor supercomputers. In contrast with traditional sequential or shared-memory supercomputers such as the Cray [9], distributed-memory machines have not been able to make good use of the HIPPI bandwidth. The network interfaces of distributed-memory machines often have low sustained bandwidth, do not perform network protocol processing, or manage connections inefficiently.

This paper focuses on one of the problems associated with I/O in distributed-memory systems: the fact that the data to be sent or received is distributed across the private memories of the nodes in the system. The problem and the approach taken are described in Section 2. Section 3 gives an overview of iWarp, the system that was used in our experiments, and Section 4 describes some commonly used data distributions. In Sections 5 and 6 the algorithms used for iWarp and their performance are presented. We end with a discussion on earlier work and conclusions.

2 Problem and Approach

Distributed-memory machines have been designed to work in a distributed fashion, and any form of serialization should be avoided whenever possible. However, network I/O is inherently a centralized, sequential activity, and it is hard to achieve high performance I/O in these architectures. This scenario poses problems in a number of areas [17]. One of them is that messages are typically distributed over the private memories of the nodes. This means that the communication software has to perform scatter and gather operations to collect or distribute the data that makes up the data stream [10]. In networking terms, this is an architecture-specific data transformation that is part of the presentation layer in the OSI layered reference model.

An additional problem is that on many of today's distributed-memory systems, internal links are slower

than the link speed of high-speed local-area networks such as HIPPI. For example, the peak throughput over each link in the Intel iWarp and Delta interconnect is 40 MBytes/second, compared to the 100 MBytes/second HIPPI link speed. To achieve high sustained network throughput, it is necessary to send data between the system and the network interface that is connected to the external network, over multiple links of the internal interconnect at the same time. This striping requirement further adds complexity to the I/O process. Striping is related to the data partitioning problem since it affects how the network interface and the distributed-memory system exchange data.

Distributed-memory machines can deal with the data distribution in a number of ways. In a first approach, array cells send data to the interface independently and the data is sorted out and grouped by the interface. This approach is simple to implement, but even if the overhead for sending and receiving blocks to or from the internal interconnect is small, the interface will become a bottleneck for fine-grain data partitionings.

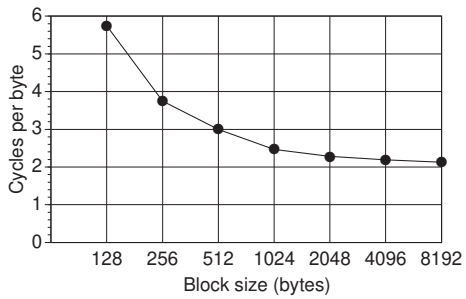

Figure 1: Cost of data movement

As an example, Figure 1 shows the overhead per byte on iWarp for sending blocks of different sizes over the internal interconnect using spooling operations [11]. The overhead consists of a fixed cost (about 120 cycles), plus a per-word cost (2 cycles per 4 bytes), and therefore the per-byte overhead for small messages is much higher than for large ones. Even though reasonable performance with small block sizes (e.g. 56 MBytes/second with a block size of 128 bytes using 4 pathways) is still achievable, the throughput is below the HIPPI link capacity.

Moreover, the network interface has to perform other tasks associated with communication (managing the network device, protocol processing, ..), and the high communication overhead associated with small blocks would consume cycles that are needed for these tasks, limiting the sustained communication throughput. On systems where communication over the internal interconnect is less efficient than on iWarp, this problem would be even more significant.

For these reasons, a different approach was used: the processor array is responsible for the scatter/gather operation. On transmit, the system constructs large messages and presents them to the network interface in an efficient way, for example, striped across multiple pathways. On receive, the system distributes the data across the processors. This approach is attractive for two reasons. First, the interface cell only has to deal with large blocks of data, independent from the data partitioning inside the array. This minimizes the cost on the network interface of exchanging data with the system. Second, distributed-memory machines typically support high-bandwidth inter-node communication, and since reshuffling parallelizes very well, many links can be used at the same time. As a result, the processor array can reshuffle data efficiently. A similar approach has been proposed for disk I/O [12].

The creation of large messages inside the distributed memory-system is done by making the interaction with the network interface a two step process. First, data is reshuffled from the distribution that was used by the application into a *canonical I/O distribution*, a mapping from which data can be sent efficiently to the network interface. Then, using this favorable distribution, the data can be sent to the network interface efficiently. The processors in the system cooperate in doing the reshuffling, thus distributing the overhead associated with the many small messages among the nodes of the system.

3 iWarp Overview

iWarp is a distributed-memory parallel computing system [4]. An iWarp cell consists of a single-chip iWarp processor and local memory. The iWarp processor integrates both a high-speed computation and communication agent in a single component. The communication agent connects the iWarp cell to four neighbors through 40 MBytes/second buses; the cells in the iWarp array are configured as a torus. The communication system supports high-speed interprocessor communication for a variety of communication models, including systolic and memory communication (e.g. conventional message passing as found in distributed memory machines) [11]. Memory communication is supported through the use of DMA-like engines that move data between the local memory and the interconnection network.

The iWarp system software includes optimizing compilers for C and FORTRAN and a cell runtime system supporting systolic and memory communication. Users can write parallel programs directly for iWarp, but for many applications, users rely on parallel program generators when using iWarp. Program generators translate a sequential user program into a program for each cell in the system, performing communication and computation concurrently on individual cells to achieve additional efficiency. Program generators can be application-specific (e.g. Apply (image processing) [13] and Assign (signal processing) [14]), or more general (e.g. HPF [15]).

iWarp systems communicate with the outside world through I/O nodes that are linked into the torus at the "border" of the array. iWarp applications perform output by sending data over the internal interconnect to the I/O node, which forwards it to an external device, such as a network or disk. Input follows the inverse path. This approach to I/O is very common, e.g. the NCube [16], the Intel iPSC [7] and Paragon machines follow the same approach.

Network Systems Corporation and Carnegie Mellon University built a HIPPI interface for iWarp (HIPPI Interface Board - HIB) [17]. The transmit and receive halves of the interface are linked into the iWarp array independently, each using 4 buses. This provides a high throughput path between the iWarp system and the HIB: 160 MBytes/second each for transmit and receive. The work presented in this paper supports high-speed I/O through the HIB.

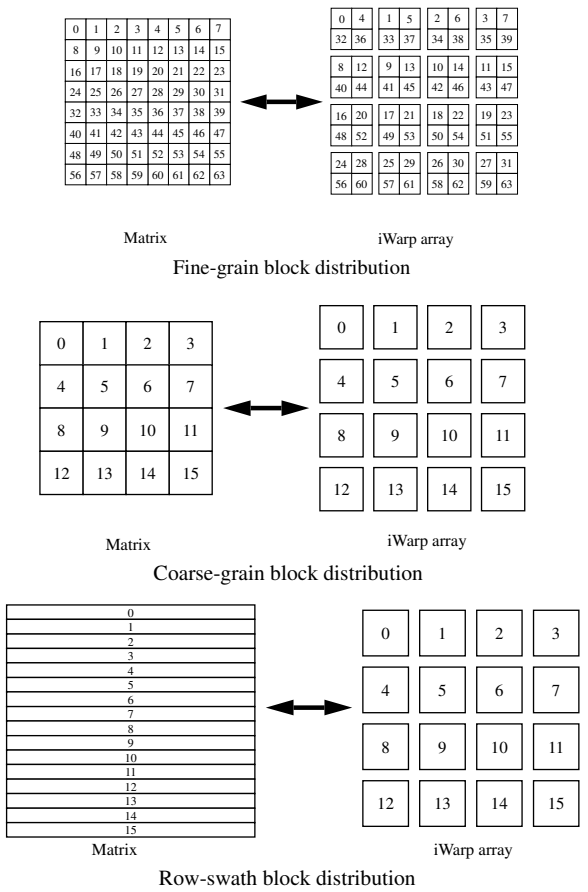

Figure 2: Common data distributions

4 Data distributions

We describe data distributions used in this paper.

4.1 Common data distributions

Figure 2 shows some of the data mappings that are used by iWarp applications. The row-swath partitioning is used by the Adapt image processing environment [18]. The coarse-grain block partitioning is used by several image processing applications and fine-grain block partitioning is used in the iWarp implementation of the LAPACK library [19]. These three examples are instances of one or two dimensional block-cyclic partitionings.

I/O of a distributed data structure becomes harder as the partitioning is finer. A good measure of the granularity is the size of blocks of data that are contiguous both on the network and in the system: when a data structure is partitioned in smaller blocks, the task of gathering the data for transmission over the network will be more significant. Table 1 shows the block size and number of blocks for the partitionings in Figure 2, given an NxN matrix mapped on a pxp processor array, using a blocking factor of b for the fine-grain mapping and assuming that the array is sent over the network in row major order. Take as an example a 1024x1024 matrix distributed over an 8x8 array. The row-swath distribution has 64 blocks of 16k elements, while the coarse-grain one has 8k blocks of 128 elements each, and a fine-grain block distribution ($b = 2$) has 16k blocks of 64 elements.

The number of different possible mappings and thus reshuffling operations needed to create appropriately sized blocks for I/O is huge. However, a small set of mappings is frequently used, and it is possible to create a data reshuffling library that reshuffles data between these mappings and the canonical I/O mapping, that is well suited for I/O. The data reshuffling library can be tuned to make optimal use of the communication

Distrib.	Type	Block size	No. blks
Row-swath	1D block	$N \times \frac{N}{p^2}$	p^2
Coarse-grain	2D block	$\frac{N}{p}$	$N \times p$
Fine-grain	2D block-cyclic	$\frac{N}{p \times b}$	$N \times p \times b$

Table 1: Data distribution complexity

resources in the distributed-memory machine. Also, note that array-level compilers that map a program onto the distributed-memory system have enough information to insert the appropriate calls into the code to do any necessary reshuffling in preparation for I/O.

4.2 Canonical I/O mapping

Both the transmit and receive halves of the HIBs have 4 physical pathways that are used to connect the HIB to the processor array. One way of streaming data from the array to the HIB is to have each column pair use one of the pathways, as is shown in Figure 3, what has the advantage that the four data streams to the HIB use different pathways.

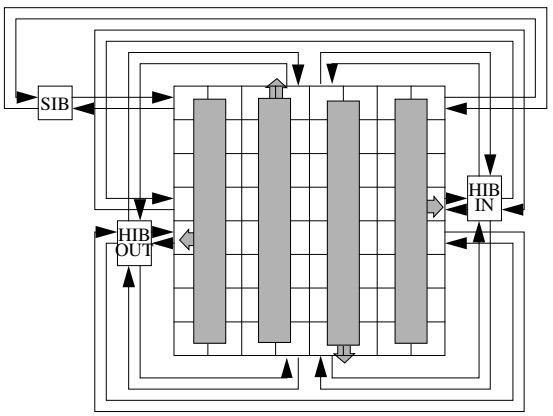

Figure 3: Data paths in the iWarp system

Figure 4 shows how data from the 8 cells in a row of the iWarp array can be merged into a contiguous stream of data by placing the blocks received from each pathway in adjacent areas of memory on the HIB. This is an efficient way for the HIB to receive data since it increases block sizes (Figure 1). In coming up with a mapping which can be efficiently handled, one has to take into account that he amount of high-speed memory on the HIB is relatively small, and the maximum message size that can be handled in a sustained fashion is 64 KBytes; this matches the maximum IP packet size. 64 KByte packets are obtained by combining 8 blocks of 8 KBytes, one from each cell in a row.

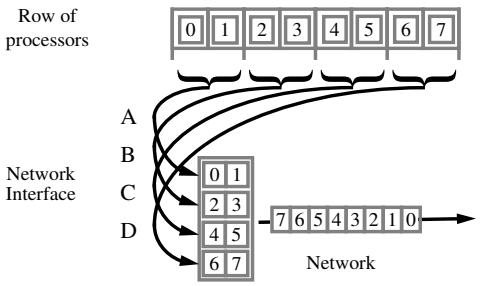

Figure 4: An iWarp row Sending data to the HIB

This mapping is only one of many possible I/O mappings. However, it is attractive since it allows efficient data exchange with the HIB and is block-cyclic, thus simplifying reshuffling for applications that use block-cyclic distributions (see below).

4.3 Block cyclic distributions

Since block cyclic distributions are commonly used, a library was implemented to provide routines that do reshuffling in support of high-speed I/O of data mapped according to a block-cyclic distribution. The characteristics of block-cyclic mappings that are supported by the algorithm are explained next.

Given a stream of bytes to be sent or received to/from the network, it can be mapped onto an iWarp system (torus) as follows: an x and y window applied to the address of the bytes in the stream of data select that bits that determine respectively in which column and row of the processor array the byte is to be placed (Figure 5). Each choice of the x and y window positions corresponds to a different block cyclic distribution. The bottom of Figure 5 illustrates how the x and y windows are selected for the three sample block cyclic distributions described earlier. The I/O mapping described in the previous section corresponds to a block-cyclic distribution with the x window starting at bit 13.

Mappings in which the x and y windows overlap correspond to mappings in which some cells do not get data. Since this is in general unattractive for applications, we will assume that the windows do not overlap. To simplify the discussion, we will also assume that the y window is to the left of the x window, and we will refer to the power of two of the number of

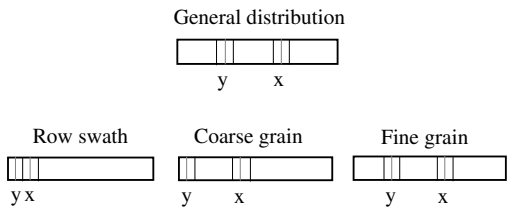

Figure 5: Bit patterns for a general distribution and for 3 common mappings

bits to the right of the x window as the *block size* of the distribution.

5 Data reshuffling algorithm

This section looks at the reshuffling algorithm for block-cyclic distributions on a 4 by 4 system, assuming that all data sizes are a power of two. Then the algorithm is extended to an 8 by 8 iWarp system, and finally non-power of two data sizes and at scaling issues are examined.

5.1 Reshuffling on a 4 by 4 array

The left hand size of Figure 6 shows one row of an iWarp array and data that is distributed across the four nodes. The blocks of data are numbered by the order in which they should be sent over the network. If each block is 16 KBytes in size, then the concatenation of a block from each column will result in a 64 KByte message on the HIB. This corresponds to the I/O mapping discussed in the previous section for a 4 by 4 system. For many applications, the block size will be less than 16 KBytes, and the efficiency of the data exchange can be improved by increasing the block size. Figure 6 shows an example of doubling the block size, which corresponds to shifting the x window that determines the mapping of the data onto the array by one bit position. Increasing the block size by a larger factor corresponds to a larger shift of the mapping window.

Figure 7 shows the data movement that is needed for the 1 bit reshuffling of Figure 6. It consists of two patterns: one moving data to the left and another moving data to the right; the two patterns are each other's mirror image. For each pattern, the cells have to support either one incoming and one outgoing stream, or one stream of data passing through the cell. Both patterns use connections in only one direction and can be executed simultaneously. Since the

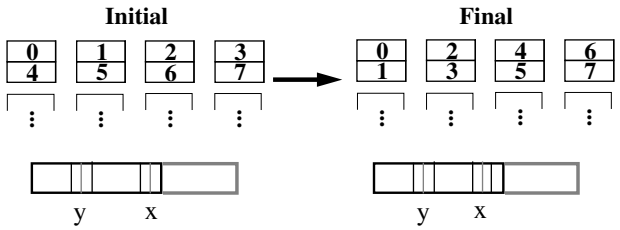

Figure 6: 1 bit shift in a 4x4 array

bandwidth of the internal links of the system is 40 MBytes/second, and the memory bandwidth (using spooling for communication) is 160 MBytes/second, this algorithm allows full utilization of the memory and link bandwidths. The cost of the reshuffling is

$$(4 \times O + \frac{B_{initial}}{T}) \times \frac{D}{B_{initial} \times 2}$$

where $B_{initial}$ is the initial block size, D is the total amount of data on each processor, T is the link throughput (40 MBytes/second for iWarp), and O is the overhead to send or receive a message.

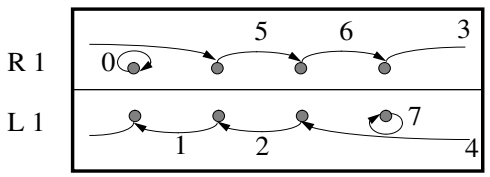

Figure 7: Patterns for a 1 bit shift in a 4x4 array

When increasing the block size by more than a factor of two, the algorithm becomes slightly more complicated. Figure 8 shows the data movement operations required for a 2 bit shift of the mapping window. It uses the same data movement patterns utilized for the 1 bit shift, plus shifted versions of these patterns. The patterns in Figure 8 have been labeled for future reference. These patterns connect each cell to all other cells in the row, so that larger bit shifts can be performed by using these same patterns repeatedly.

To perform an s-bit shift operation, each node executes the following algorithm:

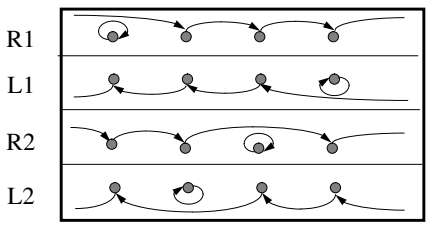

Figure 8: Patterns for a 2 bit shift in a 4x4 array

```
Alternate between phases R1/L1 and R2/L2.
For each phase do
   Calculate the address of the
   outgoing and incoming blocks
   If source = destination address then
      perform a local move and
      forward data passing by
   else
      send block in direction
      receive block from opposite direction
   Switch phases
until all data is in place
```

The calculation of the address of the outgoing and incoming blocks is based on the phase number, and the identifier of the source and destination nodes. One of them is always the node itself, while the identifier of the other node is obtained from a table.

The cost of the algorithm is the cost of a single phase times the number of phases, or

$$cost = (4 \times O + \frac{B_{initial}}{T} + PhS) \times \frac{D}{B_{final}} \times 2^{s-1}$$

where PhS is the cost of a phase switch, and s is the number of bits the window is shifted. Since $B_{final} = B_{initial} \times 2^s$, this can be rewritten as

$$cost = (2 \times O + \frac{PhS}{2}) \times \frac{D}{B_{initial}} + \frac{D}{2 \times T}$$

The cost depends on $B_{initial}$, but is independent from the s. This is to be expected: each block in the original mapping adds overhead, but the final placements of the blocks does not influence the cost.

5.2 Implementation on an 8 by 8 array

There are several ways to extend the algorithm to an 8 by 8 array. A first solution is to use a general all-to-all communication package, as is presented in [20]. Another solution is exploit the fact that our goal is to send data to the network interface, and not a general all-to-all communication. Reshuffling can be done in two steps: in the first step, all blocks are moved to the correct column pair. This is achieved essentially by applying the algorithm for the 4 by 4 array on both the odd and the even columns of the system. The final reordering of the data can be done during the actual transfer to the network interface. At that time the cells in the pairs of columns can interleave their data so that it gets sent in the appropriate order. Figure 9 illustrates it.

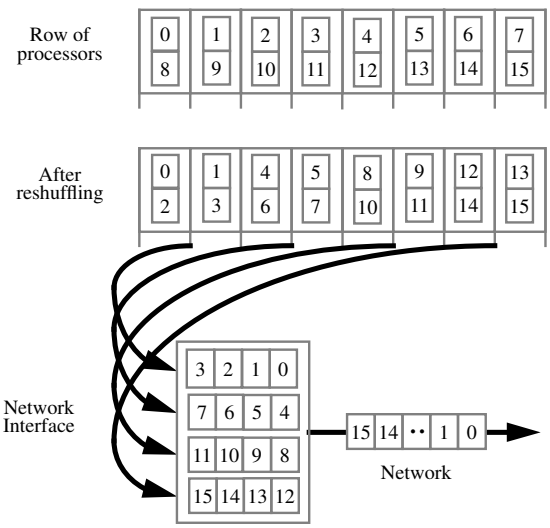

Figure 9: A simple reshuffling

When moving from a 4 by 4 to an 8 by 8 array for a fixed data size, the node memory bandwidth and CPU requirements stay the same (since they scale with the data size), but the pathway requirements increase since data has to travel across more hops. As a result, the pathways will limit the throughput on an 8 by 8 system. In order to better distribute the load on the pathways, it is advantageous to combine data movement in one direction for the even columns with data movement in the opposite direction for the odd columns (Figure 10). The labels of the data movement correspond to the labels used in the 4 by 4 array case. The algorithm presented earlier should now include the forwarding of data coming in the direction opposite to that in which the cell is sending data in that phase. The algorithm is optimal since it fully utilizes the most critical resource (the pathways).

The two approaches to reshuffling are similar in the sense that they use resources efficiently, although the

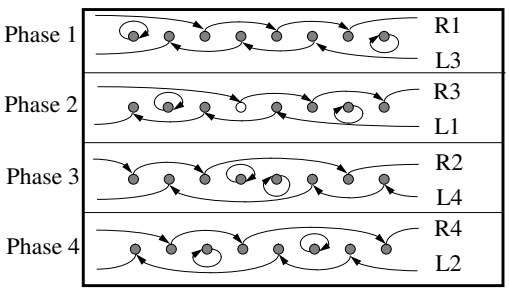

Figure 10: Patterns for a 2 bit shift in a 4x4 array

all-to-all approach described in [20] uses slightly different data movement patterns. The odd-even approach has the advantage that it has only half as many phases, and thus fewer synchronization points. The disadvantage is that it does not place the data in its final location, but this is not important for our purposes since the final ordering can be achieved efficiently during communication with the network interface.

In practice the data distribution can be in two dimensions. If reshuffling in one dimension results in blocks that are large enough to get efficient throughput to the network interface, no reshuffling has to be performed in the other dimension. For example, if the y window that is used for mapping is above bit 16, each row of the iWarp system has at least 64 KBytes of sequential data mapped on it. If the mapping is finer, reshuffling in the second dimension can be beneficial. Results presented in [20] indicate that for small block sizes, that would justify the use of 2D reshuffling, doing the reshuffling in both dimensions separately is more efficient than combining them, i.e. the algorithm above can be executed twice for those cases.

Finally, it has been assumed throughout this paper that both the data size and the system size are powers of two. For different block sizes, the same algorithm can be used as long as the ratio between the block sizes of the original and target distributions is a power of two, which is a natural choice. For different system sizes, there exist efficient communication patterns for all to all personal communication in an arbitrary system with a torus topology, as described in [20], that can be used for the reshuffling. Some additional complexity is involved in the computing of what blocks to send and receive to or from each cell.

5.3 Scaling issues

An important question is how the performance of the algorithm scales with the data size and data granularity (block size) and with the system size. The cost of the reshuffling is identical to the cost in the 4 by 4 case: twice as many nodes are used, but each node transmits and receives only at half the speed because of bandwidth limitation on the internal interconnect. For an 8 by 8 system, this formula can be used to examine the cost dependents on the application characteristics. Given a fixed block size and system size, a larger input set will correspond to more phases of the same algorithm, i.e. scaling is linear, what one would hope for. Given a certain data and system sizes, the algorithm becomes more efficient as the block size increases, which again is what would be expected.

When the system grows larger than the 8 by 8, the same algorithm can be used by applying the 4 by 4 algorithm to the appropriate columns in the system. As long as there are 4 links to the network interface, the same mechanisms apply. As the system gets bigger, the pathways will become more of a bottleneck compared with the memory bandwidth, since data has to travel across more hops. Ultimately, the bisection bandwidth of the system will limit the performance. The reshuffling bandwidth should clearly increase linearly with the increase of system size in the direction orthogonal to the one in which the reshuffling is taking place. Given that the network has a fixed bandwidth, reshuffling will become less of an issue as the system grows.

6 Performance measurements

Experiments were performed on an 8x8 iWarp system to assert the viability of the approach taken.

An 8 MByte data structure was stored on a 64-node iWarp system following 2-dimensional block-cyclic distributions. In the y dimension 64 KByte blocks were stored according to a block-cyclic distribution, i.e., each row in the iWarp system stored 64 KBytes of contiguous data. In the x dimension, block-cyclic distributions with block sizes ranging from 128 to 8192 bytes (of contiguous data in each cell) were used, allowing the study of the impact of different distribution granularities.

In a first experiment the performance of the reshuffling code was evaluated by itself. Two different implementations were used: in the "slow" version, the computation agent is directly involved in the transfer of all the data, while in the "fast" version of the algorithm the communication agent is responsible for the forwarding of data going through a cell, whenever it is destined to some other cell. Although the performances differ considerably, they are both suitable for

our purposes, being faster than the ultimate bottleneck in the system, the HIPPI network.

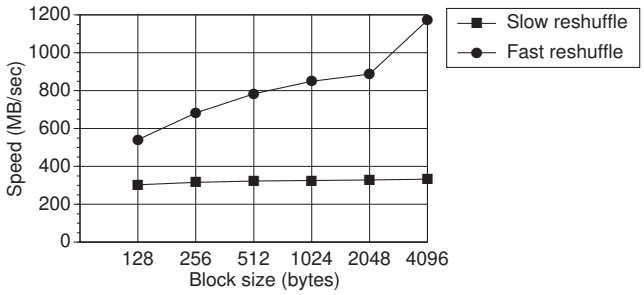

Figure 11: Performance of the Reshuffling Algorithm

A number of things can be observed by examining the performance of the algorithm as a whole, in Figure 12. First, notice that the algorithm performs well, and can provide higher throughput to the HIB than what can be sustained over a HIPPI connection, with either implementation of the reshuffling algorithm ("slow" and "fast" versions). Also, a comparison of these results with the performance achieved by the code with no reshuffling, shows that the reshuffling results in some slowdown. This is caused by two main factors: the reshuffling code does have to be executed; and when both parts of the algorithm are being executed, they have to compete for the internal network resources, and there is contention between some of the logical paths used. Finally, notice that the version of the code that is slower in reshuffling the data ends up overall being faster than the other implementation. That can be easily explained by the already mentioned contention for resources, which increases when a faster reshuffling algorithm is used, thus resulting in poorer performance. Interestingly, the use of optimal libraries to separately implement the two phases of the algorithm would not achieve optimum performance. Also, note that no reshuffling is performed in the 8 KByte block case.

Figure 13 gives the breakdown of the time spent in the algorithm when using the "slow" version of the code. The results presented here are what is experienced by cells in the last row of the iWarp system, those cells being the last group to finish executing the algorithm. Each node performs a reshuffling operation. Then, the cells in the first row of the system start sending their data, while the nodes in the remaining rows wait for their time to send data to HIB; once an iWarp cell is done it starts a new sequence of reshuffling / waiting / sending to HIB, until all the data has been sent. This allows the reshuffling in one row to overlap with the sending of data in other

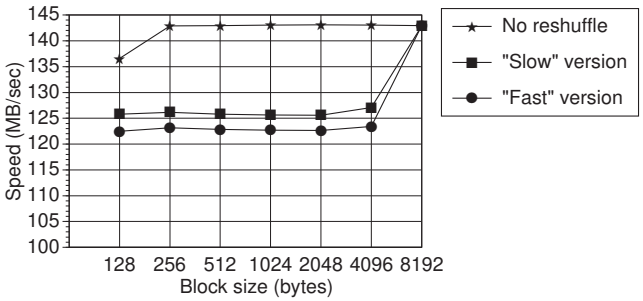

Figure 12: Throughput to the HIB

rows, so that the total execution time of the reshuffling phase is not that critical for the overall performance of the algorithm. As can be seen, there is a substantial amount of waiting between the reshuffling and the time at which data can actually be sent to the HIB, during which cells in other columns are sending their data.

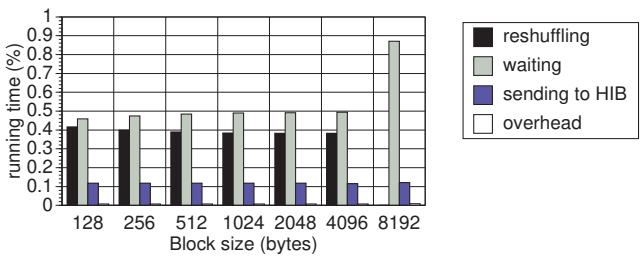

Figure 13: Breakdown of the algorithm execution time

Finally, we look at the impact of using an appropriate I/O mapping. The block size in the I/O mapping was reduced by a factor of two, i.e. set to 4 KByte blocks in each cell. The performance with this I/O mapping is compared with the performance using the original one in Figure 14. Some loss of performance is observable and caused by the fact that, in this case among other things, the reshuffling and dispatching phases could not overlap as much, due to a shorter dispatch operation. These results are what one would expect: a smaller block size in the canonical distribution results in a less efficient data transfer. Even though the slow down is relatively small, it is important to realize that the overhead on the HIB has increased, since it has to receive twice as many blocks from the internal interconnect. This can have a negative impact on the other tasks that have to be performed by the HIB.

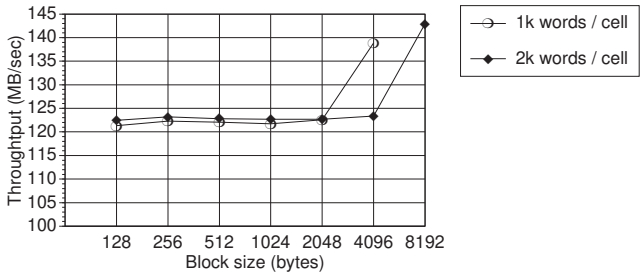

Figure 14: Comparison between a 4K and an 8K canonical distribution

7 Related Work

Kwan and Terstriep [21] show how the data distribution and data representation on the CM2 is a significant hurdle when doing I/O over the HIPPI interface, and how it limits the throughput. This paper shows how this problem can be addressed on iWarp. The iWarp interconnect has enough bandwidth and flexibility to support high-speed reshuffling, as shown.

In [12] the authors an analogous problem when dealing with file I/O in the Intel Touchstone Delta. They also observed that in order to achieve good performance it is important to send large blocks of data to the file servers that are part of the system. Although their solution strategy is similar, the work differs in a number of ways. First, our target throughput is much higher. Second, we explore the regularity of the distributions to achieve good reshuffling performance; this is important, given our performance goals. Finally, the flexibility of the routing in iWarp plays an important role in our algorithms, since it allows the use of independent paths to increase bandwidth. For example, using x-then-y routing, a common routing strategy on tightly-coupled distributed-memory systems such as the Intel Delta, it would not be possible to use the 4 links to transfer data from the system to the HIPPI.

8 Conclusion

The data distribution is an important problem that has to be solved to achieve high-speed I/O on distributed-memory systems. This paper has shown that it is advantageous to have the system create large blocks of data that can be handled efficiently by the network interface. The high bandwidth of the interconnect, and the opportunities for parallelism in the reshuffling algorithm result in efficient reshuffling and increased throughput. The measurements show this library easily supports the HIPPI bandwidth.

Acknowledgements

We would like to thank various members of the iWarp and Nectar groups at CMU, and in special we would like to thank Professor Thomas Gross for his advice, and Bwolen Yang, Susan Hinrichs and Michael Hemy for their invaluable help.

References

[1] ANSI, "High-performance parallel interface - mechanical, electrical and signalling protocol specification (hippi-ph)." ANSI X3.183-1991, 1991.

[2] M. de Prycker, "Asynchronous transfer mode." Ellis Harwood, 1991.

[3] ANSI, "Fibre channel physical and signaling interface (fc-ph)." Working draft proposed American National Standard for Information Systems, 1992.

[4] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An integrated solution to high-speed parallel computing," in *Proceedings of Supercomputing '88*, (Orlando, Florida), pp. 330–339, IEEE Computer Society and ACM SIGARCH, November 1988.

[5] A. J. G. Hey, "Supercomputing with transputers - past, present and future," in *Proceedings of the 1990 International Conference on Supercomputing*, pp. 479–489, ACM, June 1990.

[6] K. Kaneko, M. Nakajima, Y. Nakakura, J. Nishikawa, I. Okabayashi, and H. Kadota, "Processing element design for a parallel computer," *IEEE Micro*, vol. 10, pp. 26–38, April 1990.

[7] S. L. Lillevik, "The touchstone 30 gigaflop delta prototype," in *Proceedings of the Sixth Distributed Memory Computing Conference*, pp. 671–677, IEEE, April 1991.

[8] L. W. Tucker and G. G. Robertson, "Architecture and applications of the connection machine," *IEEE Computer*, vol. 21, pp. 26–38, August 1988.

[9] A. Nicholson, J. Golio, D. A. Borman, J. Young, and W. Roiger, "High speed networking at cray research," *Computer Communication Review*, vol. 21, pp. 99–110, January 1991.

[10] T. Gross and P. Steenkiste, "Architecture implications of high-speed i/o for distributed-memory computers." In preparation, 1993.

[11] S. Borkar, R. Cohn, G. Cox, T. Gross, H. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb, "Supporting systolic and memory communication in iwarp," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, (Seattle), pp. 70–81, ACM/IEEE, May 1990. Also published as CMU Technical Report CMU-CS-90-197.

[12] J. M. d. R. Rajesh Bordawekar and A. Choudhary, "Design and evaluation of primitives for parallel i/o," in *Proceedings of Supercomputing '93*, (Oregon), pp. 452–461, ACM/IEEE, November 1993.

[13] B. Baxter and B. Greer, "Apply: A parallel compiler on iwarp^{TM} for image-processing applications," in *Proceedings of the Sixth Distributed Memory Computing Conference*, pp. 186–193, IEEE, April 1991.

[14] D. R. O'Hallaron, "The assign parallel program generator," in *Proceedings of the Sixth Distributed Memory Computing Conference*, pp. 178–185, IEEE, April 1991.

[15] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross, "Programming task and data parallelism on a multicomputer," in *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, p. to appear, May 1993.

[16] J. P. Hayes, T. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "A microprocessor-based hypercube supercomputer," *IEEE Micro*, vol. 6, pp. 6–17, October 1986.

[17] P. A. Steenkiste, M. Hemy, T. Mummert, and B. Zill, "Architecture and evaluation of a high-speed networking subsystem for distributed-memory systems," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, p. To appear, IEEE, May 1994.

[18] J. A. Webb, "Architecture-independent global image processing," in *10th International Conference on Pattern Recognition*, (Atlantic City, NJ), pp. 623–628, IEEE, June 1990.

[19] H. T. Kung and J. Subhlok, "A new approach for automatic parallelization of blocked linear algebra computations," in *Proceedings of Supercomputing '91*, pp. 122–129, ACM, November 1991.

[20] S. Hinrichs, C. Kosak, D. O'Hallaron, T. M. Stricker, and R. Take, "An architecture for optimal all-to-all personalized communication." Personal Communication, Feb. 1994.

[21] T. T. Kwan and J. A. Terstriep, "Experiments with a gigabit neuroscience application on the cm-2," in *Proceedings of Supercomputing '93*, (Oregon), pp. 133–142, ACM/IEEE, November 1993.