Architecture and Evaluation of a High-Speed Networking Subsystem for Distributed-Memory Systems

Peter Steenkiste, Michael Hemy, Todd Mummert, Brian Zill

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891

Abstract

Achieving high-speed network I/O on distributed-memory systems is difficult because their architecture is in general ill-suited for communication processing. Some of the common problems are: inability to do protocol processing, inefficient handling of data distribution, and poor management of the I/O. In this paper we present an I/O architecture that addresses these problems and supports high-speed network I/O on distributed-memory systems. The key to good performance is to partition the work appropriately between the system and the network interface. We perform some communication tasks on the distributed-memory parallel system since it is more powerful, and less likely to become a bottleneck than the network interface. Tasks that do not parallelize well are performed on the network interface and hardware support is provided for the most time-critical operations. We emphasize the use of simple I/O mechanisms that can be used by programming tools that map applications on the distributed-memory system to implement efficient I/O for the class of applications they support.

This architecture has been implemented for the iWarp distributed-memory system. We describe this implementation and present performance results.

1. Introduction

Supercomputer applications have to communicate over a high-speed network, for example, to 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 [12] have become very popular in supercomputer centers, and all commercially available supercomputers have a HIPPI interface. HIPPI supports a data rate of 800 Mbit/second or 1.6 Gbit/second. In addition to HIPPI, there are a number of high-speed network standards in various stages of development by standards bodies. These include ATM (Asynchronous Transfer Mode) [6] and Fibre Channel [26].

Meanwhile, distributed-memory computer systems [2, 11, 13, 15, 24] are becoming the architecture of choice for many supercomputer applications. The reason is that they are inherently scalable, and provide relatively inexpensive computing cycles compared with traditional uniprocessor or shared-memory multiprocessor supercomputers. However, while traditional sequential or shared-memory supercomputers such as the Cray have been able to make good use of the HIPPI bandwidth [16], distributed-memory machines have been much less successful. The network interfaces of distributed-memory machines often have low sustained bandwidth, do not perform network protocol processing, or manage connections inefficiently. High-speed I/O for distributed-memory machines is difficult because the entire system, including the architecture, software, programming model and applications, have been tuned to work optimally in a distributed fashion, while connecting to a network is an inherently centralized and sequential activity.

A simple approach to network I/O on distributed-memory machines is to make the network interface responsible for all network-related processing, so the structure of the network interface will be similar to that of a sequential system. However, this places a heavy burden on the network interface since it has to support the I/O of a large number (potentially thousands) of processors. The network interface can easily become a bottleneck (top Figure 1), resulting in poor performance.

Another approach is to provide a much simpler network interface and to minimize the amount of work that is assigned to it by performing some of the communication tasks on the distributed-memory system itself (bottom Figure 1). Earlier research [20] shows that the time spent on sending and receiving network data is distributed over several operations such as copying data, buffer management, protocol processing, and interrupt handling, and different overheads dominate depending on the circumstances (e.g. packet size). By executing some operations on the distributed-memory system, we can reduce the communication software bottleneck on the interface. In this paper we present an I/O architecture that

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract number MDA972-90-C-0035, in part by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

1. Transport and network layer protocol processing is performed on the network interface, but hardware support is provided for time-critical tasks such as data checksumming (Section 4). The reason is that protocol processing does not parallelize well.
2. The network interface presents the user with a simple I/O model that can be implemented efficiently. This model, called the streams interface, allows applications on the distributed-memory system to directly manage multiple connections to the outside world (Section 5).
3. It is the responsibility of the distributed-memory system to combine the data that is distributed over the private memories of the compute nodes in large blocks that can be handled efficiently by the network interface, and to do the reverse scatter operation on receive (Section 6). The reason is that the creation of messages, i.e. reshuffling of data, is a highly parallel operation that can be performed efficiently on distributed-memory systems.

We claim that this distribution of tasks between the network interface and the system allows us to support high-speed I/O in a cost effective way.

One of the characteristics of the above mapping of tasks is that it scales appropriately with increased data complexity and network speed. Protocol processing, the overhead of which is proportional to the network speed, is performed on the network interface, while data management, the complexity of which depends on the application, is done on the same system that performs the application.

Applications on distributed systems can be very diverse with respect to I/O. For example, they use different data distributions, and have to interact with a wide range of external devices (displays, disk arrays, other computers). As a result of this diversity, general-purpose solutions to I/O are likely to be inefficient, both in terms of performance and resource utilization. Instead, we are providing a number of simple mechanisms to deal with the communication processing problems and applications can use these mechanisms to implement I/O efficiently. Distributed-memory systems are often programmed using programming tools that map a sequential program on the system. These tools manage the resources in the system based on an understanding of the characteristics of the application and distributed-memory system, and they are in the best position to manage the I/O using the mechanisms described in this paper.

This architecture has been implemented for the iWarp system and a HIPPI network by Network Systems Corporation (NSC). The transmit board has been in use by iWarp applications to send data to a HIPPI framebuffer since spring 93. The full interface was delivered in early 1994. UDP over IP is operational, and the TCP port is nearing completion. NSC also built an interface for DEC workstations based on the same architecture as the iWarp interface; it connects to the TURBOchannel [7].

3. iWarp overview

iWarp is a distributed-memory parallel computing system [2]. An iWarp cell consists of a single-chip iWarp processor and a local memory. The iWarp processor integrates both a high-speed computation and communication agent in a single component. The communication agent connect the iWarp cell to four neighbors through 40 MByte/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 communication and memory communication (e.g. conventional message passing as found in distributed memory machines) [3]. 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) [9] and Assign (signal processing) [17]), or more general (e.g. HPF [23]).

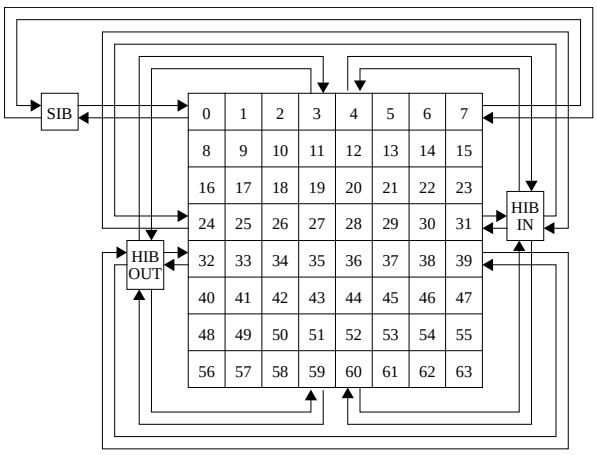


Figure 3: Connection of interface cells to iWarp array

iWarp systems communicate with the outside world through I/O nodes that are linked into the torus at the ‘‘edge’’ of the array. Figure 3 shows an example of an I/O node (Sun Interface Board, or SIB) linked into the array in one dimension, and two I/O nodes (HIPPI Interface Boards, or HIB) that are linked into the array using 4 buses. iWarp applications perform output by

sending data over the internal interconnect to the I/O node, which forwards it to the 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 [10], and the Intel iPSC [15] and Paragon machines follow the same approach.

4. Transport protocol processing

Transport protocol processing (e.g. TCP or UDP over IP) is one of the potential bottlenecks for high speed network I/O for distributed-memory systems, and it is tempting to try to distribute this operation. However, this is very difficult since most of the protocol processing functions are inherently sequential. There is potential for parallelism between transmit and receive processing (if one is transmitting and receiving at the same time), and ACK and data processing can sometimes proceed in parallel [18], but overall, useful parallelism is limited. For these reasons, it is desirable to have protocol processing performed in a central location, i.e. the network interface. A number of distributed systems use a similar approach [19].

One protocol processing task that does parallelize well is the checksum calculation for the Internet protocols, and it could be performed efficiently on the distributed-memory system. However, calculating the checksum on the system results in an odd ordering of the checksum calculation relative to protocol processing. Ideally, the checksum calculation on transmit should take place after the protocol has broken up the data stream in packets, and on receive, the calculation should take place before protocol processing is invoked since corrupted packets have to be ignored. Performing the checksum calculation on the system results in checksumming being performed too early on transmit and too late on receive. Although it is possible to deal with this, it would add complexity to the software, and for this reason we perform the checksum calculation on the network interface.

The operations associated with protocol processing fall in two categories: overheads associated with every packet sent over the network and overhead that scales with the number of bytes sent. As networks get faster, the per-byte overheads, i.e. data copying and checksumming, become the dominating overheads, both because the other overheads are amortized over larger packets and because these operations make heavy use of a critical resource: the memory bus. The key to making these operations efficient is to streamline the dataflow [22], i.e. avoid unnecessary copying of data, and to calculate the checksum while data is being copied. The per-packet overhead includes time spent in the TCP/IP code, plus overhead associated with buffer management, interrupt handling, and context switching. Careful implementation of these operations, as discussed below, should make their cost acceptable. Based on these observations, the network interface for iWarp relies on iWarp processors to do most of the transport protocol processing, but provides hardware support for per-byte operations.

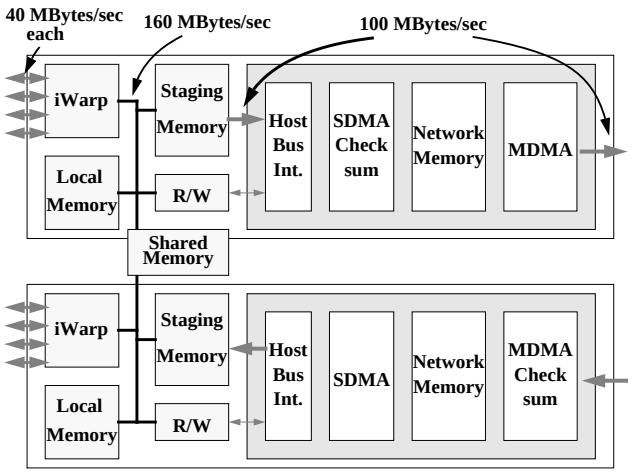


Figure 4: HIB block diagram

Figure 4 shows the architecture of the iWarp-HIPPI network interface, or HIPPI Interface Board (HIB). It consists of a transmit and a receive half. Each half includes an iWarp processor that links the network interface into the iWarp torus. The iWarp processor performs protocol processing, while a ‘‘communication acceleration block’’ (CAB - shaded area in Figure 4)) provides support for critical protocol processing operation: data transfer, checksumming and buffering. The architecture uses two iWarp processors instead of one because of data bandwidth requirements. The critical resource in the architecture is the memory bus of the iWarp processors since all data that is sent/received has to flow over it, and the bus is also used for program and local data accesses. Using two iWarp processors instead of one doubles the bandwidth available for these operations from 160KByte/sec to 320KByte/sec.

The data flow on transmit is as follows: the distributed-memory system forwards the data over the iWarp torus to the network interface and it is placed in staging memory using the iWarp DMA engines. Once a block of data is ready to be transmitted, the iWarp processor issues a ‘‘send’’ call similar to the Unix socket ‘‘write’’ call. This invokes the TCP/IP processing code which performs packetization, and asks the CAB to DMA the data into network memory. The checksum is performed in hardware as part of the data transfer. The packets remain in network memory until they have been acknowledged by the destination, so the CAB functions as a retransmit buffer. The data flow on receive is exactly the inverse, and network memory holds incoming data until the application on the distributed-memory system is ready to receive it.

The staging memories are dual-port RAMs with a bandwidth of 160MByte/second on the iWarp side and 100MByte/second on the network side. The main role of

the staging memories in the architecture is to efficiently gather data coming from the different iWarp buses (transmit) or to scatter data (receive). Since the DMA engines on the iWarp chip interleave the data on the iWarp memory bus in small blocks (8 bytes), it is necessary to use static RAM to achieve high throughput. As a result, the staging memories are small: 128KByte for each direction.

The operation of the network interface is similar to that of a sequential system, except that the data source and sink is the distributed-memory system, and not the iWarp processors on the interface. The iWarp nodes on the network interface effectively represent the iWarp array on the HIPPI network.

The implementation of the TCP/IP protocol stack for iWarp differs from a traditional workstation implementation in a number of ways. The most obvious difference is that processing is distributed over two processors, so the transmit and receive components of TCP/IP have to be separated. A shared memory (Figure 4) allows the two components to keep a consistent protocol state. Second, the protocol stack has to be modified to make use of the outboard storage and checksum calculation. These changes are similar to the protocol stack modifications needed on workstations that have outboard storage and checksum support; they are briefly discussed in [21].

Finally, iWarp cells have a very minimal runtime system, and several facilities that are available in most operating systems, such as buffer management, and that are needed by protocols, are not available. This increases the implementation effort since these facilities have to implemented. However, it has the advantage that we can use optimized implementations, and can avoid a lot of the overhead introduced in protocol implementations in existing operating systems. By using an optimized UDP/TCP/IP implementation [5] and efficient buffer management, we are able to keep the overhead of protocol processing within acceptable bounds.

Figure 5 shows the bandwidth we can achieve from the interface board to the network. Measurements for raw HIPPI and UDP over IP give the same throughput results, i.e. the UDP/IP implementation is very efficient. The main difference between the two cases is that the idle time on the iWarp cell is lower in the UDP case (40%-56%) than in the raw HIPPI case (74%-79%). The current bottleneck in the system is the microcode on the CAB: it limits us to sending about 3000 packets per second. We expect this number to go up after optimization.

5. The streams I/O model

The transfer of data between the application on the system and the network is a two phase process. In a first phase (transmit), data is transferred from the system to the interface cell, and in the second phase, the data is sent

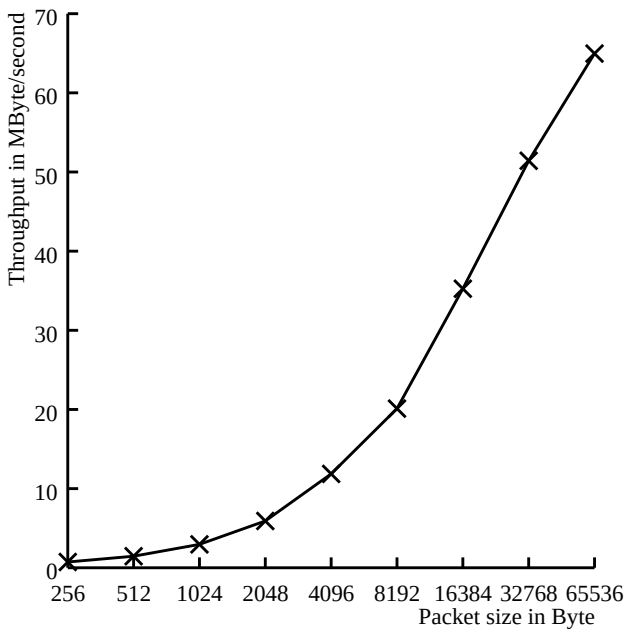


Figure 5: Bandwidth from network interface to network as a function of packet size

over the network. The data transfer over the network is controlled by the communication protocols, as described above. However, the timing of the data transfers between the distributed-memory system and the interface cell is critical to the performance of the application, and should be under the control of the application. For example, incoming data should not be transferred to the system before the application is ready to use it since it will compete with the application for bandwidth on the internal interconnect. It also has to be possible to control the relative order of data transfers over different connections. If the order is different from what the application expects, this could result in inefficiency (e.g. an extra buffering step), or in the extreme case, in deadlock.

Managing the I/O is mainly a resource management problem: both the distributed-memory system and the network interface have limited resources (memory, buses, ..), and how they are allocated to support I/O will have a significant impact on performance. On sequential systems, this task is traditionally performed by the operating system. On distributed-memory systems, there are a number of ways to handle this task.

A first solution is to have the I/O managed on the front-end of the distributed-memory system, typically a Unix workstation: it sets up each communication operation on the network interface. This approach is attractive since the management is done on a general-purpose system, but it has the disadvantage that it adds overhead, since every communication operation has to go through the front end. This will only result in high throughput for very large data transfers. This approach is used for the HIPPI interface of

the CM2.

A more attractive solution is to have the transfers controlled directly by the system. However, providing a general-purpose I/O function is both undesirable and unnecessary. First, applications on distributed-memory systems have very diverse I/O requirements. The requirements differ in how much data is communicated, the data format and the timing requirements. Providing a general-purpose I/O function, specifically for exchanging data between the system and the interface, is likely to either have low performance or to consume too many resources. For example, a general-purpose I/O function could use links in the internal interconnect that are already heavily used by the application, while other links are idle. Second, distributed-memory machines often are programmed using higher-level programming tools that map a sequential program onto the system. These tools control the resources, since their understanding of both the application and system characteristics allows them to use the resources efficiently, and they are the natural entity to control I/O.

In our approach, the network interface implements a simple I/O model, called the streams model, and the application code executing on the system uses this model to control I/O. The application is responsible for the data transfer between the array and the HIB. A *stream* is a sequence of bytes that is sent from the iWarp torus to a single network destination, or from a single network source to the iWarp torus. Applications will in general require multiple streams, and the HIB can support multiple streams, either simultaneously, or interleaved.

The stream interface allows the programmer (or programming tool) to define a stream. This definition includes the source or destination on the network, the protocol used (TCP or UDP over IP, or raw HIPPI), plus the definition of one or more data transfers. A data transfer is defined by:

- The format of the data on the iWarp side. This is defined in the form of an application data unit (ADU), a block of data that is contiguous in the data sequence on the network, and that is also communicated with the iWarp array as a unit. ADUs can be striped across multiple pathways to achieve a network throughput higher than the internal link speed of the distributed-memory system. The specification of an ADU includes the block size and pathway(s) used.
- The format of the data on the network side. If the reliable stream protocol is used (TCP/IP), the data format on the network side is a byte stream, and no format specification is needed; packetization and header creation are done by TCP. If the datagram protocol (UDP/IP) or raw HIPPI is used, then the user is responsible for packetization, and the user has to specify the message size and (optionally) a header for network communication.
- Mapping between the format of the data stream over the network and iWarp. This corresponds to specifying the order in which data travels over the network.

The stream manager, a system program running on the HIB, uses the format information to manage the data flow between the iWarp torus and the network connection as efficiently as possible. Applications can supply the specification of data transactions over the streams at application startup time or at runtime.

Figure 6 shows an example of a single stream that uses a single iWarp pathway. Blocks of data (ADUs) are transferred from the pathway into staging memory, from where they are transferred over the network using the CAB. In the figure, ADUs are transferred through staging memory (using double buffering - not shown) one at a time. If ADUs are small enough, the stream manager will transfer multiple ADUs at a time to minimize overhead. For example, with 16KByte ADUs, 4 ADUs will be transferred at a time resulting in 64KByte writes to the network; with double buffering this would utilize the entire 128KByte staging memory.

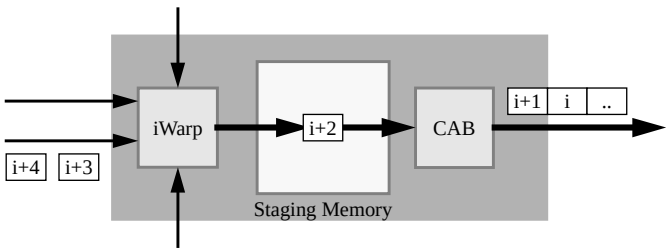


Figure 6: Example of single stream using one pathway

Figure 7 shows an example of a single stream that uses four pathways. The stream manager builds ADUs in staging memory by DMAing 1 block from 4 different pathways. The data organization on the iWarp side is clearly more complicated (see next section) but this approach makes it possible to achieve throughputs higher than the internal link speed. Again, the stream manager will try to group ADUs to improve performance.

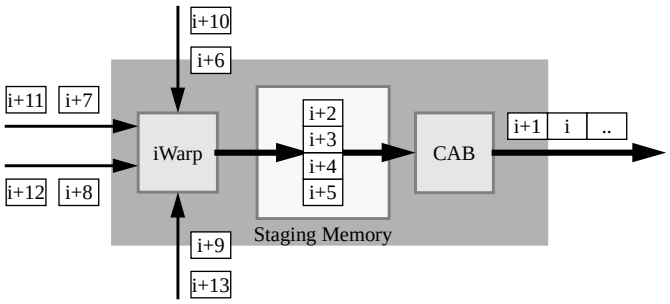


Figure 7: Single stream striped across 4 pathways

Applications will sometimes need several streams. Typically there will not be enough resources on the HIB to have all streams operate efficiently at the same time, either because there are not enough DMA engines in the HIB iWarp cell, or because the staging memory would become too fragmented. The solution is to have only a

small number (possibly one) of *active streams*. These streams have resources allocated to them, and the stream manager controls their data flow. The other streams are idle, waiting for resources. The unit of interleaving, i.e. how long a stream remains active, is defined by the application, and will typically have some relationship to the structure of the application data.

6. Data reshuffling in support of high-speed I/O

To efficiently utilize the large number of processors in a distributed-memory computer, applications typically use data parallelism. Data is partitioned into equal-sized blocks, which are distributed across the processors, and each processor operates on the data that is assigned to it. Both the type and granularity of data partitionings varies widely between application. As a result, I/O operations include an extensive scatter/gather operation that is application specific. The scatter/gather operation also has to deal with striping the data stream over multiple buses to achieve higher throughput, as was discussed in the previous section.

In this section we first describe some commonly used data partitionings, and we then discuss how data partitioning can be handled efficiently.

6.1. Data partitioning

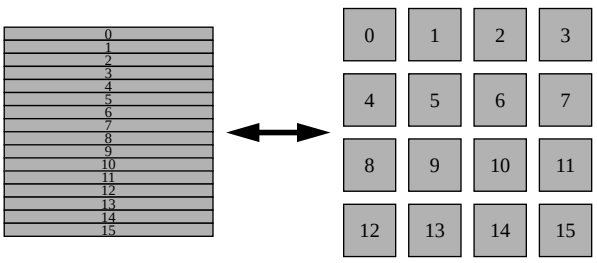
Figure 8 shows some of the mappings that are used by iWarp applications. The row-swath partitioning is used by the Adapt image processing environment [25]. The coarse-grain block partitioning is used by several image processing applications and fine-grain block partitioning is used in the iWarp implementation of the LAPACK library [14]. All 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 of Figure 8 for an $N \times N$ matrix mapped on a $p \times p$ processor array, using a blocking factor of b for the fine-grain mapping; we assume the array is sent over the network in row order. A specific example is given in Table 2. We see that the block sizes are quite small for the 2D partitionings.

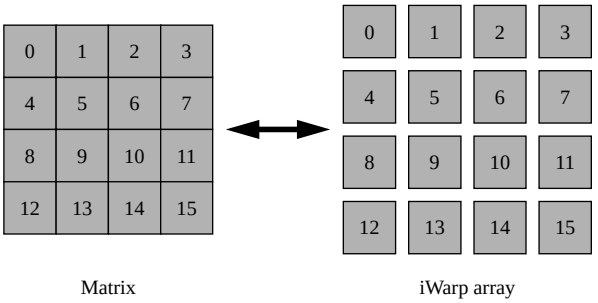
6.2. Strategy

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.

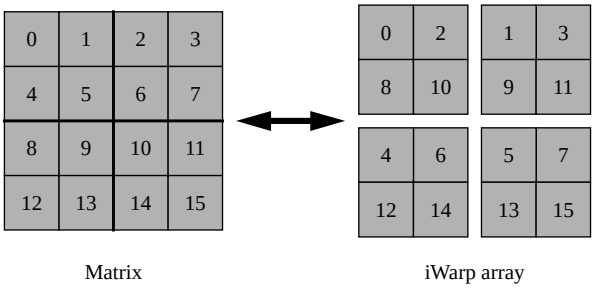
In the approach we selected, the processor array is responsible for the scatter/gather operation. On transmit, it constructs large messages and presents them to the



(a) Row-swath partitioning over 4 by 4 array



(b) Coarse-grain block partitioning over 4 by 4 array



(c) Fine-grain block partitioning over 2 by 2 array

Figure 8: Common data mappings on iWarp

Name	Type	Block Size	Number of Blocks
Row-swath	1D block	$N \times \frac{N}{p^2}$	p^2
Coarse-grain block	2D block	$\frac{N}{p}$	$N \times p$
Fine-grain block	2D block-cyclic	$\frac{N}{p \times b}$	$N \times p \times b$

Table 1 Granularity of 3 common mappings

Name	Block Size (elements)	Number of Blocks
Row-swath	16K	64
Coarse-grain block	128	8K
Fine-grain block	64	16K

Table 2 Block counts and sizes for N=1024, p=8, and b=2

interface in an efficient way, for example striped across multiple pathways. On receive, it 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 [1].

The number of different mappings, and thus reshuffling operations needed to create appropriately sized blocks, is potentially unlimited. However, a small set of mappings is frequently used, and it is possible to create a data reshuffling library that reshuffles between these mappings and an *I/O mapping*, a mapping that is well suited for I/O. The data reshuffling library can be optimized to make optimal use of the communication resources in the distributed-memory machine. Note that array-level compilers that map a program onto the distributed-memory system know about the mapping of the data structures, understand the architecture of the system (e.g. number of buses), and can insert the appropriate calls to do any reshuffling in preparation of I/O.

6.3. Performance

We developed a library that deals with block-cyclic mappings, which includes the mappings of Section 6.1 [4]. The library takes as input a data structure distributed using a block-cyclic mapping, for example the coarse-grain block mapping of Figure 8b, and reorganizes the data in contiguous blocks of 8KByte that are distributed as show in Figure 9: adjacent blocks are in adjacent cells. Once the data is organized following this I/O mapping, each column pair can send the data in parallel to the HIB over the connections shown in Figure 10, where there are placed contiguously in staging memory to form a contiguous 64KByte message. This is the most efficient way for the HIB to receive data.

Figure 11 shows the throughput we can achieve between the iWarp system and the HIB for different message sizes and when different numbers of buses are used, under optimal conditions, i.e. the data is stored in the cells labeled A-D in Figure 10. We see how we have to use

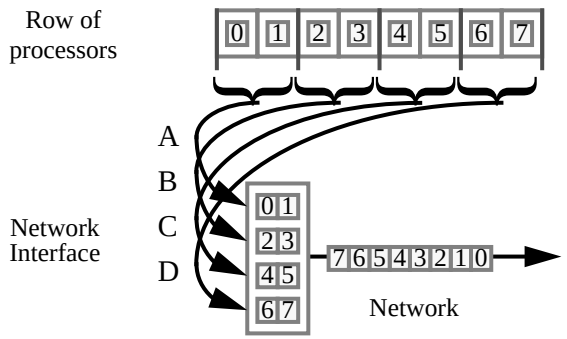


Figure 9: Example of an I/O mapping

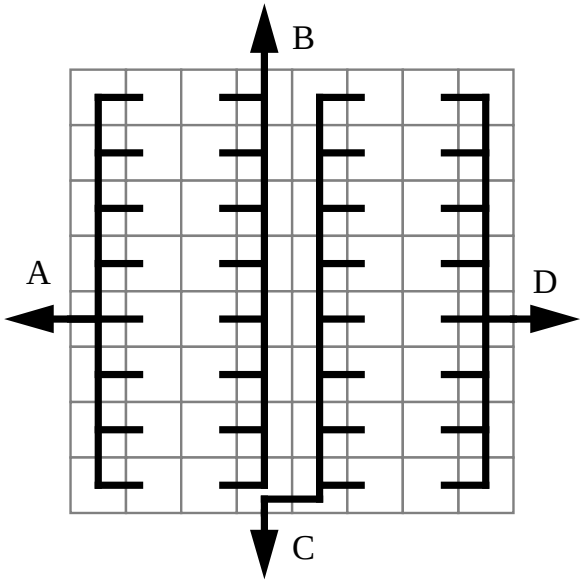


Figure 10: Connections used for I/O

multiple pathways to achieve HIPPI-like throughput. For example for the above example of 64KByte messages, we achieve a throughput of over 140 MByte/second using four pathways. Note that in the case of 2 and 4 pathways, the size of the messages transferred over each pathway are respectively one half and one fourth of the total message size.

Figure 12 shows the bandwidth between the iWarp system and the HIB for different partitioning granularities the x-axis gives block size of the partitioned data structure. The measurements include the cost of reshuffling the data into the I/O mapping of figure 9, and of transferring the data to the network interface. The numbers are for a coarse-grain block partitioning of a 512KByte data structure, but other block-cyclic partitionings give similar results. We see how reshuffling allows us to achieve high throughput, even for the very fine partitionings. One of the reasons for the good performance is that some of the reshuffling can be done in parallel with the transfer to the HIB.

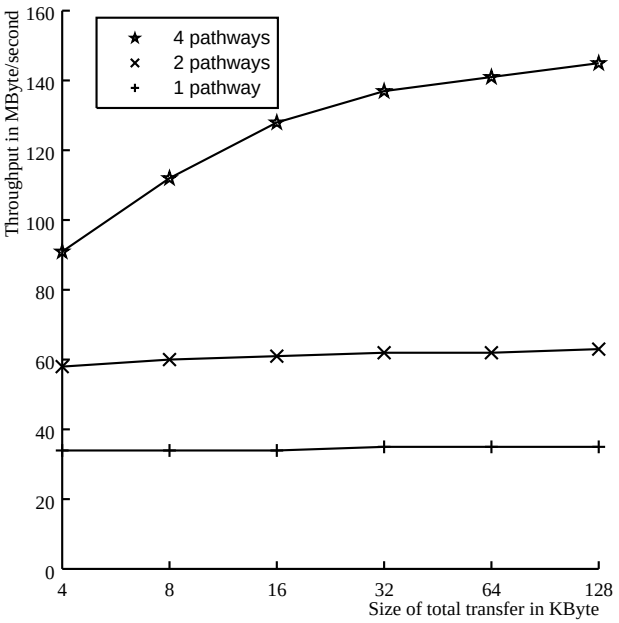


Figure 11: Raw bandwidth between the iWarp array and the network interface

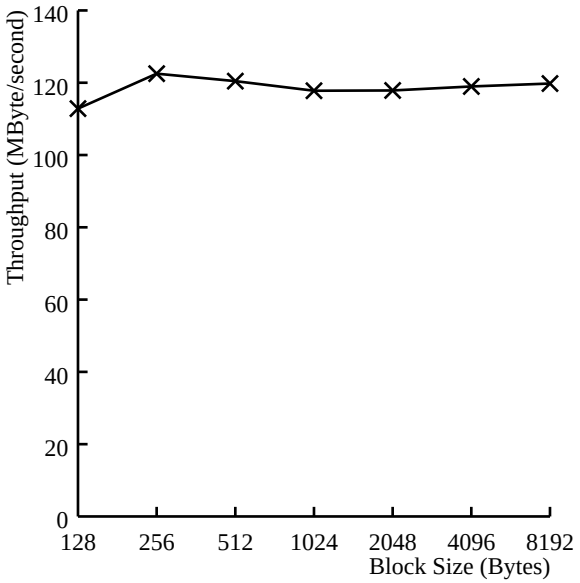


Figure 12: Throughput between the iWarp array and the network interface for a distributed data structure

In order to evaluate the end-end path between the iWarp system and the network, we created a minimal iWarp application that creates simple images and sends them to the HIPPI framebuffer. We observed a sustained throughput of 53 MByte/second. This number includes minimal processing on the system, striping across 4 pathways, and communication processing for raw HIPPI and framebuffer control on the network interface.

7. Conclusion

Achieving high-speed network I/O on distributed-memory systems is difficult because their architectures is in general ill-suited for communication processing. Some of the common problems are: inability to do protocol processing, inefficient handling of distribution of data in the array, and management of the I/O. In this paper we presented an I/O architecture that supports high-speed network I/O. We perform several communication tasks on the distributed-memory system since it is more powerful than the network interface, and thus less likely to become a bottleneck. Specifically, management of the distributed data is performed on the system, while protocol processing is performed on the network interface, with hardware support for memory intensize operations, i.e. checksumming and data movement. Management of I/O is the shared responsibility of the application on the array and system software on the network interface.

We emphasize the use of simple I/O mechanisms that can be used by programming tools that map applications on the distributed-memory system to implement efficient I/O for the class of applications they support. Examples are the stream concept and the reshuffling libraries. General-purpose complete I/O solutions are likely to be inefficient, given the wide diversity of applications in distributed-memory machines.

We described an implementation of this I/O architecture for the iWarp distributed-memory system. Our performance results show that the architecture supports high-speed network I/O.

Acknowledgements

We would like to acknowledge the contributions of the iWarp group at CMU in integrating the HIB with the iWarp system. Tom Warfel, Michael Gillinov and Thomas Gross were especially helpful. Terry Allen and Claudson Bornstein wrote different versions of the reshuffling library. Several people at NSC, including Paul Georgeson, Clayton Haapala, Tom Lane, Gary Mullenberg, and Jeff Schmitt, deserve credit for implementing the HIB hardware and microcode.

References

1. Rajesh Bordawekar, Juan Miguel del Rosario and Alok Choudhary. Design and Evaluation of Primitives for Parallel I/O. Proceedings of Supercomputing '93, Oregon, November, 1993, pp. 452-461.

2. Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin,

P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. Proceedings of Supercomputing ’88, IEEE Computer Society and ACM SIGARCH, Orlando, Florida, November, 1988, pp. 330-339.

3. Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H.T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. Supporting Systolic and Memory Communication in iWarp. Proceedings of the 17th Annual International Symposium on Computer Architecture, ACM/IEEE, Seattle, May, 1990, pp. 70-81.

4. Claudson Bornstein and Peter Steenkiste. Data Reshuffling in Support of Fast I/O for Distributed-Memory Machines. In preparation.

5. David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. "An Analysis of TCP Processing Overhead". *IEEE Communications Magazine 27*, 6 (June 1989), 23-29.

6. M. de Prycker. *Asynchronous Transfer Mode.* Ellis Harwood, 1991.

7. *TURBOchannel Overview.* Digital Equipment Corporation, 1990.

8. Thomas Gross and Peter Steenkiste. Architecture Implications of High-Speed I/O for Distributed-Memory Computers. In preparation.

9. Leonard Hamey, John Webb, and I-Chen Wu. Low-Level Vision on Warp and the Apply Programming Model. In *Parallel Computation and Computers for Artificial Intelligence*, J. Kowalik, Ed., Kluwer Academic Publishers, 1987, pp. 185-199.

10. John P. Hayes, Trevor Mudge, Quentin F. Stout, Stephen Colley and John Palmer. "A Microprocessor-based Hypercube Supercomputer". *IEEE Micro 6*, 5 (October 1986), 6-17.

11. Anthony J. G. Hey. Supercomputing with Transputers - Past, Present and Future. Proceedings of the 1990 International Conference on Supercomputing, June, 1990, pp. 479-489.

12. ANSI. High-Performance Parallel Interface - Mechanical, Electrical and Signalling Protocol Specification (HIPPI-PH). ANSI X3.183-1991.

13. Kaneko, K., Nakajima, M., Nakakura, Y., Nishikawa, J., Okabayashi, I. and Kadota, H. "Processing Element Design for a Parallel Computer". *IEEE Micro 10*, 2 (April 1990), 26-38.

14. H. T. Kung and Jaspal Subhlok. A New Approach to Automatic Parallelization of Blocked Linear Algebra Computations. Proceedings of Supercomputing ’91, IEEE, Albequerque, November, 1991, pp. 122-129.

15. Sigurd L. Lillevik. The Touchstone 30 Gigaflop DELTA Prototype. Proceedings of the Sixth Distributed Memory Computing Conference, April, 1991, pp. 671-677.

16. Andy Nicholson, Joe Golio, David A. Borman, Jeff Young, and Wayne Roiger. "High Speed Networking at Cray Research". *Computer Communication Review 21*, 1 (January 1991), 99-110.

17. David R. O’Hallaron. The Assign Parallel Program Generator. Proceedings of the Sixth Distributed Memory Computing Conference, IEEE, April, 1991, pp. 178-185.

18. Erich Rutche and Matthas Kaiserswerth. TCP/IP on the Parallel Protocol Engine. Proceedings of the 4th IFIP Conference on High Performance Networks, IFIP, Liege, Belgium, December, 1992, pp. C2 1-16.

19. Raj K. Singh, Stephen G. Tell, Shaun J. Bharrat, David Becker, and Vernon L. Chi. A Programmable HIPPI Interface for a Graphics Supercomputer. Proceedings of Supercomputing ’93, Oregon, November, 1993, pp. 124-132.

20. Peter Steenkiste. Analyzing Communication Latency using the Nectar Communication Processor. Proceedings of the SIGCOMM ’92 Symposium on Communications Architectures and Protocols, ACM, Baltimore, August, 1992, pp. 199-209.

21. Peter A. Steenkiste, Brian D. Zill, H.T. Kung, Steven J. Schlick, Jim Hughes, Bob Kowalski, and John Mullaney. A Host Interface Architecture for High-Speed Networks. Proceedings of the 4th IFIP Conference on High Performance Networks, IFIP, Liege, Belgium, December, 1992, pp. A3 1-16.

22. Peter A. Steenkiste. "A Systematic Approach to Host Interface Design for High-Speed Networks". *IEEE Computer 26*, 3 (March 1994), to appear.

23. Jaspal Subhlok, Jim Stichnoth, Dave O’Hallaron, and Thomas Gross. Programming Task and Data Parallelism on a Multicomputer. Proc. of the ACM Symposium on Principles and Practice of Parallel Programming, May, 1993, pp. 13-22.

24. Lewis W. Tucker and George G. Robertson. "Architecture and Applications of the Connection Machine". *IEEE Computer 21*, 8 (August 1988), 26-38.

25. Jon A. Webb. Architecture-Independent Global Image Processing. 10th International Conference on Pattern Recognition, IEEE, Atlantic City, NJ, June, 1990, pp. 623-628.

26. X3T9 - I/O Interface. *Fibre Channel Physical and Signaling Interface (FC-PH).* Rev. 2.2 edition, 1992. Working draft proposed American National Standard for Information Systems.