A High-Speed Network Interface for Distributed-Memory Systems: Architecture and Applications

Peter Steenkiste
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

October 9, 1996

Abstract

Distributed-memory systems have traditionally had great difficulty performing network I/O at rates proportional to their computational power. The problem is that the network interface has to support network I/O for a supercomputer, using computational and memory bandwidth resources similar to those of a workstation. As a result, the network interface becomes a bottleneck. 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. Some communication tasks are performed 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. This architecture has been implemented for the iWarp distributed-memory system and has been used by a number of applications. We describe this implementation, present performance results, and use application examples to validate the main features of the I/O architecture.

To appear in ACM Transactions on Computer Systems.

This research was supported by the Advanced Research Projects Agency/CSTO monitored by the Space and Naval Warfare Systems Command under contract N00039-93-C-0152.

Peter Steenkiste can be reached at steenkiste@cs.cmu.edu. His mailing address is School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213.

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 [2] have become very popular in supercomputer centers, and all commercially available supercomputers provide 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) [16] and Fibre Channel [26].

Meanwhile, distributed-memory computer systems [6, 23, 27, 32, 44] 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 [33], 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, resulting in poor performance.

Another approach is to provide a 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. Earlier research [40] 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 takes this approach: it supports high-speed I/O using a simple network interface in a cost effective way. This architecture has been implemented for the iWarp distributed-memory system and we present performance results for this implementation throughout the paper.

The remainder of this paper is organized as follows. In Section 2 we describe our architecture for I/O in distributed-memory systems. We then give a brief overview of iWarp (Section 3) and we discuss our implementation of the different components of the I/O architecture for iWarp: protocol processing (Section 4), session management (Section 5), and support for data distribution (Section 6). In Section 7 we describe a number of applications that use the iWarp HIPPI interface and we discuss how they use the features of the architecture. Finally, we present related work in Section 8, and briefly look at how the I/O architecture applies to other distributed-memory systems 9. We summarize in Section 10.

2 I/O architecture

We describe the challenges associated with network I/O on distributed-memory systems and we present an I/O architecture that addresses these challenges.

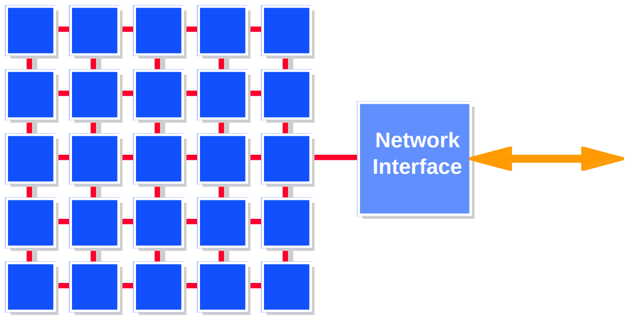


Figure 1: Connecting a distributed-memory system to a network

2.1 Challenges

Distributed-memory systems communicate over an external network (e.g. HIPPI) through a *network interface* node that is connected both to the external network and the internal interconnect of the system (Figure 1). The role of the network interface is to forward data between the internal and external network. High-speed I/O for distributed-memory machines is difficult because the entire system, including the architecture, software, programming model and applications, have been designed to work in a distributed fashion, and any form of serialization is avoided if possible. The connection to a network, however, is inherently centralized, creating a potential bottleneck. Specifically, the following three I/O tasks can be hard to implement:

1. Distributed-memory machines achieve their power by distributing the work over a large number of relatively slow processors, but *communication protocol processing* does not parallelize well.

2. The applications often have to manage multiple connections, and this involves *scheduling resources* in both the distributed-memory system and on the network interface. This is a complicated task and there is a conflict between using general-purpose solutions and providing mechanisms that are tailored to a specific application.

3. Data that is sent or received over the network is 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 [20]. In networking terms, this is an architecture-specific data transformation that is part of the presentation layer.

The three processing tasks that are hard to implement efficiently for distributed-memory systems correspond roughly to the transport, session and presentation layers of the OSI network model (Figure 2).

An additional problem is that on some 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 MByte/second, compared to the 100 MByte/second HIPPI link speed. To achieve high sustained network throughput, it is necessary to send data between the system and the network interface over multiple links at the same time. This striping requirement further adds complexity to the I/O process.

Applications on distributed systems have very diverse I/O requirements. For example, they use different data distributions and 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 need a set of simple mechanisms that can be used to implement efficient, customized communication support for a wide range of applications. One of the motivations for this approach is that distributed-memory systems are increasingly being programmed using programming tools that map a sequential program on the system. These tools manage the resources in the

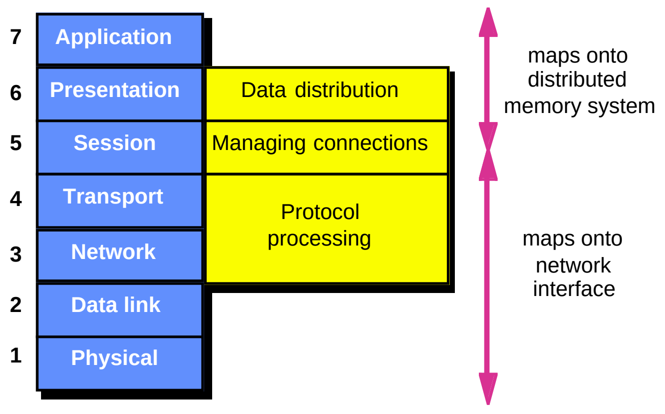


Figure 2: Mapping of protocol stack

system based on an understanding of the characteristics of the application and distributed-memory system and they are in the best position to also optimize the I/O. This however requires an I/O architecture that gives higher level software sufficient control over how I/O is performed.

2.2 Architecture

In our I/O architecture, as many communication tasks as possible are performed on the distributed-memory system, and the network interface is a relatively simple system that is optimized to deal with tasks that do not parallelize well. Moreover, the functions allocated to the distributed memory system are performed in close cooperation with the application, allowing optimization across the application and I/O operations. Specifically, we map the communication tasks in the following way:

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 interface, called the streams interface[1], that can be used by the application code to control I/O, that can be implemented efficiently and that 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.

An important feature of this mapping of tasks is that it scales appropriately with increased network speed and data complexity. The overhead of protocol processing is proportional to the network speed and this task is performed on the network interface, which is typically developed for a specific network technology and speed. The complexity of data management, on the other hand, depends strongly on the application, and this task is performed on the distributed-memory system, which also executes the application.

[1]The stream interface has no relationship to the streams interface found in some versions of the Unix operating system.

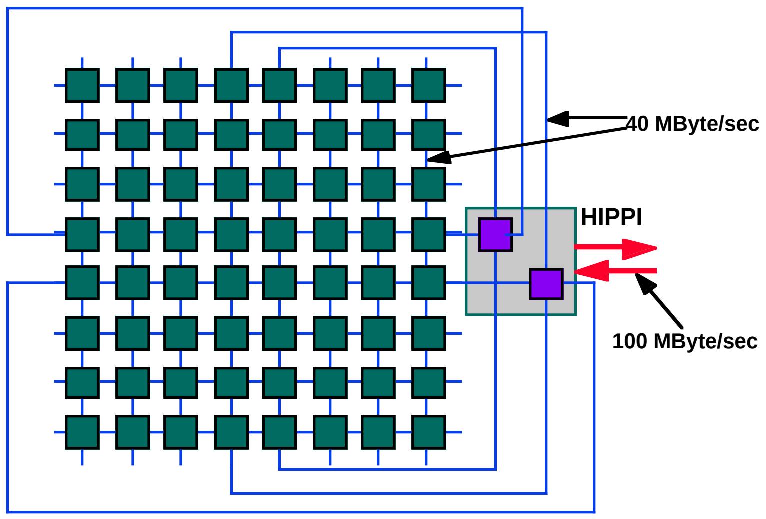


Figure 3: Connection of HIPPI network interface to iWarp distributed-memory system

This architecture has been implemented for the iWarp system and a HIPPI network. The HIPPI Interface Board (HIB) was built by Network Systems Corporation (NSC) based on a joint CMU/NSC design. The transmit board has been in use by iWarp applications since Spring 93 and the full interface was delivered to CMU in 1994. The network interface architecture described in this paper supports high-speed I/O in a cost effective way for iWarp. We have measured sustained throughputs of 55 MByte/second for simple iWarp applications sending images to a HIPPI framebuffer, and 40 MByte/second for more complex iWarp programs sending data to the Cray C90 at the Pittsburgh Supercomputer Center as part of a heterogeneous distributed computing application. Some of the applications that use the HIPPI interface are described in Section 7.

3 iWarp overview

iWarp is a distributed-memory parallel computing system [6]. 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 connects 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 [7]. In systolic communication, the CPU writes data directly onto the interconnect, thus minimizing communication latency. Memory communication is supported through the use of spools, on-chip DMA 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 parallelizing compilers when using iWarp. The parallelizing compilers 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) [21] and Assign (signal processing) [36]), or more general (e.g. the Fx parallelizing FORTRAN compiler [19]).

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 the example of a HIPPI interface connected to the iWarp distributed-memory system through eight links of the internal iWarp torus; four links are used for transmit and four for receive. 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 [22], and the Intel iPSC [32] and Paragon [25] machines follow the same approach.

4 Transport protocol processing

Protocol processing (e.g. TCP or UDP over IP) is one of the potential bottlenecks in network communication. In this section we describe how the iWarp HIPPI interface supports protocol processing.

4.1 High-level design

While it is tempting to try to distribute protocol processing, this is difficult since most of the protocol processing functions are inherently sequential. There is potential 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 [37], 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 [39].

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. Unfortunately, calculating the checksum on the system results in an odd ordering of the checksum calculation relative to protocol processing. Normally, the checksum calculation on transmit takes place after the protocol has broken up the data stream in packets. On receive, the calculation takes place before protocol processing is invoked since corrupted packets have to be ignored. Performing the checksum calculation on the distributed-memory system is too early on transmit and too late on receive. Although it is possible to deal with this, it would add considerable complexity to the software. Specifically, on transmit, the system can calculate the checksum on small blocks of data, e.g. 8-16 KByte, and the network interface can then calculate the per-packet checksum cheaply by combining the partial checksums; more expensive adjustments will be needed if packet boundaries do not coincide with block boundaries. On receive, protocol processing can be delayed until after the data has been transferred to the distributed-memory system; this might influence protocol performance since it will delay the acknowledgment of data. To avoid this complexity, we perform the checksum calculation on the network interface.

The operations associated with protocol processing fall in two categories: overhead 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 flow of data during I/O [41], so that the number of times that the data is touched is minimized. Example optimizations include the elimination of redundant data copy operations and the calculation of 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, can 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.

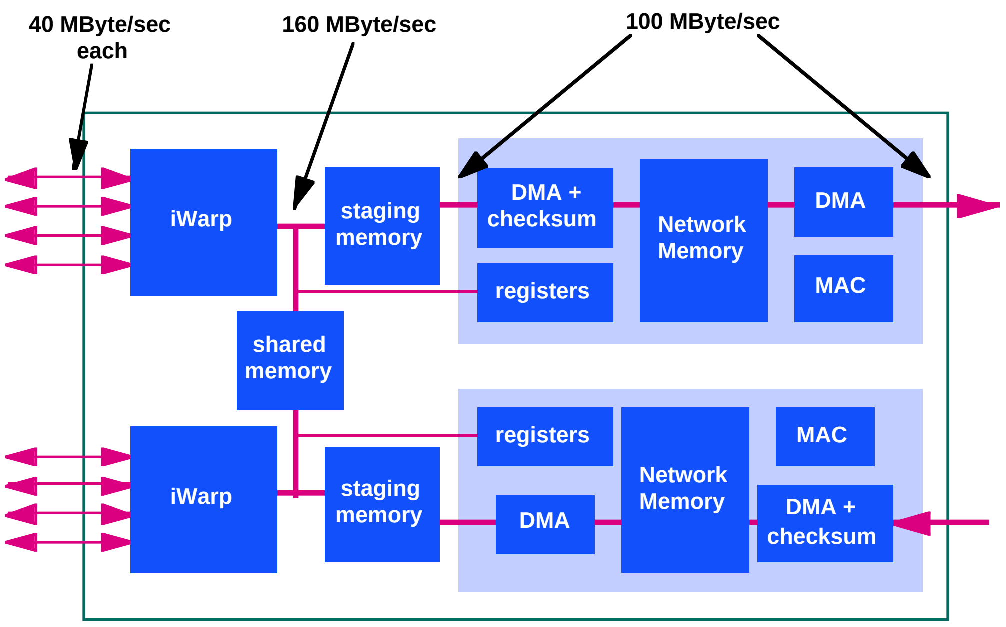


Figure 4: HIPPI Interface Board (HIB) architecture

4.2 Network interface architecture

Figure 4 shows the architecture of the iWarp-HIPPI network interface, or HIPPI Interface Board (HIB). It consists of a transmit and a receive pipe. Each pipe includes an iWarp processor that links the network interface into the iWarp torus, a staging memory for short-term packet storage, and a "communication acceleration block" (CAB - shaded area in Figure 4). The iWarp processor is responsible for per-packet operations, such as protocol processing, while the CAB provides support for per-byte operations: data transfer, checksumming and buffering. The CAB architecture used on the HIB is similar to the Gigabit Nectar workstation CAB [42, 28], which provides support for per-byte operations for network communication on workstations. 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, not the iWarp processors on the interface. The iWarp nodes on the network interface effectively represent the iWarp array on the HIPPI network.

The data flow on transmit is as follows: the distributed-memory system forwards the data over the iWarp torus to the network interface where 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 and it is inserted in the packet header by the CAB. 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. A more detailed description of the CAB can be found in [42].

A critical requirements for high-bandwidth communication is a high memory bandwidth on the I/O node. The bandwidth has to be high enough to support the data stream to/from the network, the data stream to/from the internal interconnect, accesses to the data stream for protocol processing (e.g. checksumming), and program and data accesses as part of regular program execution. Note that, especially if striping is used, the aggregate stream of accesses will have low locality in memory, making it harder to support the required bandwidth. These high bandwidth requirements typically make the memory bus the bottleneck of the network interface.

The HIB architecture uses two mechanisms to optimize memory bandwidth utilization. First, the HIB uses two iWarp processors instead of one, thus doubling the memory bandwidth available for these operations from 160 MByte/sec to 320 MByte/sec. Second, the HIB supports the data stream using two levels of memory, each optimized for different types of accesses. We use a small dual-ported static RAM (staging memory) for short-term storage; it can support at high rates the scatter/gather operations resulting from striping. This memory is backed up by a larger dynamic RAM (network memory); data transfers between staging and network memory, and network memory and the network are performed in large contiguous data blocks. The IP checksum is calculated in hardware during these transfers, thus eliminating an access to the data by the CPU. With this memory organization the outgoing or incoming data streams have to cross the iWarp memory bus only once, which is the minimum load possible. The other load on the bus consist of accesses to the program and data structures, which are placed in a separate memory (not shown in Figure 4), similar to that found on compute nodes.

The memory design described above is in part driven by iWarp-specific features. For example, transfers between memory and the iWarp interconnect go through the iWarp chip and the DMA engines on the iWarp chip interleave the data streams on the iWarp memory bus at a fine granularity (8 bytes). As a result, the bandwidth requirements could only be met by using static RAM on iWarp memory bus, resulting in the small (128KByte for each direction) staging memories. However, optimizing the use of the memory bus is critical for any high-throughput network interface, and the use of a Communication Acceleration Block that provides buffering and checksumming is an effective way of reducing the load memory bus. This has been demonstrated not only with the HIB for distributed-memory systems, but also for workstations [28, 14]. As a result we expect that the overall HIB architecture is applicable to other distributed-memory systems.

4.3 Protocol software

The implementation of the TCP/IP protocol stack for iWarp differs from a traditional workstation implementation in three ways: 1) protocol processing is distributed over two processors, 2) TCP has to make use of outboard buffering and checksum support, and 3) the stack runs outside of a traditional OS environment.

The most obvious difference is that processing is distributed over two processors. A shared memory (Figure 4) allows the two components to keep a consistent protocol state. Most of the protocol functions fall under either transmit or receive processing. The transmit half is responsible for sending packets with user data and acknowledgments or window updates. The receive half receives packets and updates the protocol state based on the information in the packet header of incoming packets, i.e. acknowledgments and window updates. If the new protocol state could trigger the transmission of a packet, the receive half interrupts the transmit half. The final main protocol function is the handling of slow and fast TCP timeouts. This function could potentially be implemented on either the transmit or receive half. We decided to handle timeouts on the transmit half, both because it is typically less loaded and because timeouts most often result in transmit activity.

Second, the protocol stack has to be modified to make use of the outboard buffering and checksum calculation. These changes are similar to the protocol stack modifications needed on workstations that use a network adapter with outboard buffering and checksum support. The idea is that instead of passing a copy of the data through the protocol stack and having each layer operate on the data, a descriptor of the data is passed through the stack and all data-touching operations are combined and performed in hardware (Figure 5). On transmit, a descriptor describing the location of the data in staging memory travels down the protocol stack and the TCP checkpointing code adds information to the descriptor where the checksum calculation should start and where the final checksum should be placed in the packet header. The driver uses the information in the descriptor to generate a request for the CAB to DMA the data from staging memory to network while performing the checksum calculation. On receive, the checksum is calculated

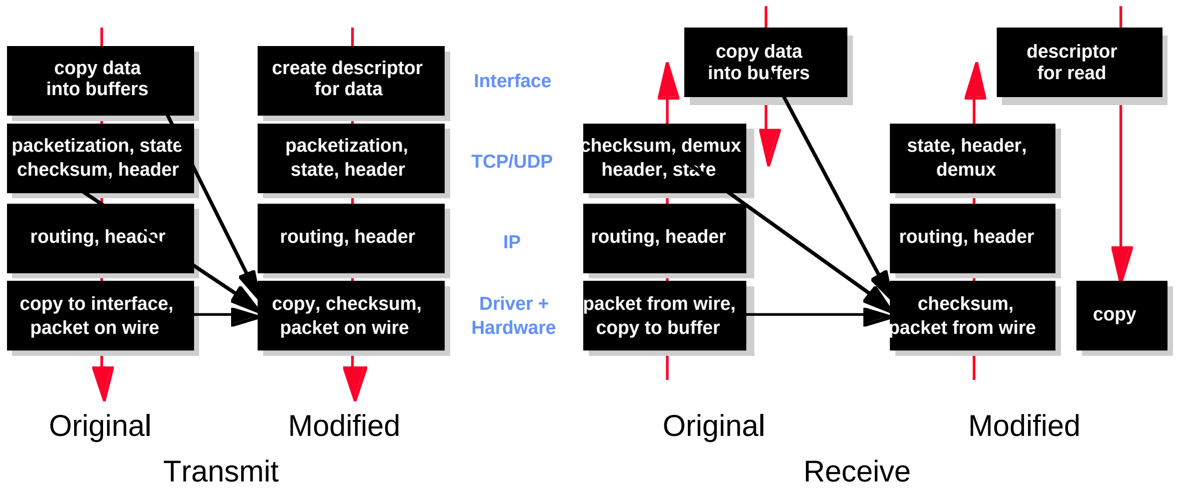

copy data into buffers
packetization, state checksum, header
routing, header
copy to interface, packet on wire
create descriptor for data
packetization, state, header
routing, header
copy, checksum, packet on wire
Interface
TCP/UDP
IP
Driver + Hardware
copy data into buffers
checksum, demux header, state
routing, header
packet from wire, copy to buffer
descriptor for read
state, header, demux
routing, header
checksum, packet from wire
copy
Original
Modified
Transmit
Original
Modified
Receive

Figure 5: Stack changes to use outboard buffering and checksum support

while the data flows from the network into network memory. The driver creates a descriptor for each packet, includes the checksum, and passes it up the protocol stack. The TCP code performs protocol processing using the hardware-calculated checksum and places the descriptor on the receive queue, making it available for transfer into the distributed-memory system. A more detailed discussion on the use of outboard buffering and checksumming, as it was implemented in a Unix protocol stack, can be found in [28].

Finally, iWarp nodes have a minimal runtime system that lacks support for utilities such as buffer management and timeouts, functions that are supported by full-fledged operating systems. Implementing these utilities increases the implementation effort, but it has the advantage that we can optimize and customize these modules and avoid a lot of the overhead present in existing operating systems. For example, instead of using mbufs, the standard package to manage kernel buffers in Unix, we developed buffer management software that is optimized for representing outboard buffers and for supporting outboard checksumming. The stream manager (Section 5) and the protocol stack share this buffer management package, thus minimizing data passing overhead for send and receive operations. We also use an optimized send/receive interface (instead of a standard interface such as Berkeley sockets), e.g. it supports asynchronous sends and posting of receives that allow the stream manager to efficient overlap send/receives with processing and interactions with the distributed-memory system. We also optimized the UDP/TCP/IP implementation itself using standard techniques such as header prediction [10]. The above optimizations keep the communication overhead within acceptable bounds. On a DEC Alpha workstation 3000/400, it takes about 300 microseconds from the time a user-level application issues a small write until a packet send request is handed to the network adapter, using the DEC OSF1 implementation of TCP/IP. The equivalent number for the iWarp protocol stack (measured on the same platform) is less than 50 microseconds.

Figure 6 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 node 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. The TCP/IP is operational in an emulation environment on DEC Alpha workstations.

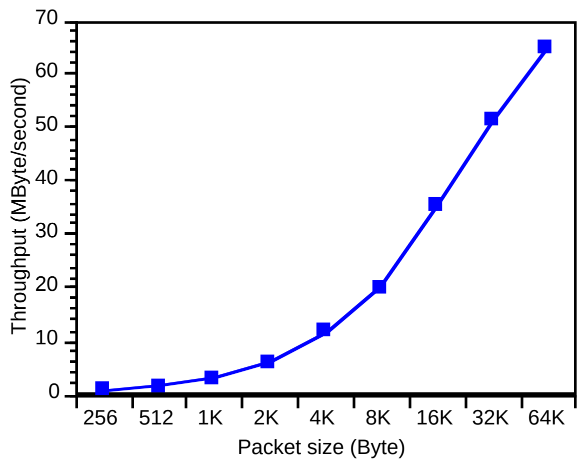


Figure 6: Throughput from HIB to network

5 The streams I/O architecture

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 distributed-memory system to the network interface, and in the second phase, the data is sent over the network. The data transfer over the network is controlled by the communication protocols, as described above. Managing the first phase is mainly a resource management problem: both the distributed-memory system and the network interface have limited resources (e.g., memory, link bandwidth), 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. However, applications on distributed-memory systems have very diverse I/O requirements, and we can identify the following requirements for the data movement between the application buffers on the distributed-memory system and the system buffers in the network interface:

- Support high bandwidth communication efficiently for both regular and irregular distributions data distributions.
- Provide an application I/O interface that is independent from the protocols or devices used on the external network.
- Support striping across multiple links to the network interface for applications requiring a network bandwidth greater than the internal link bandwidth.
- Allow applications to manage communication, e.g. specify when to receive data or request to interleave data across multiple connections.

5.1 Architecture

One approach to the resource management problem 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. The streams architecture provides a more attractive solution: transfers are controlled directly by the application running on the distributed memory system.

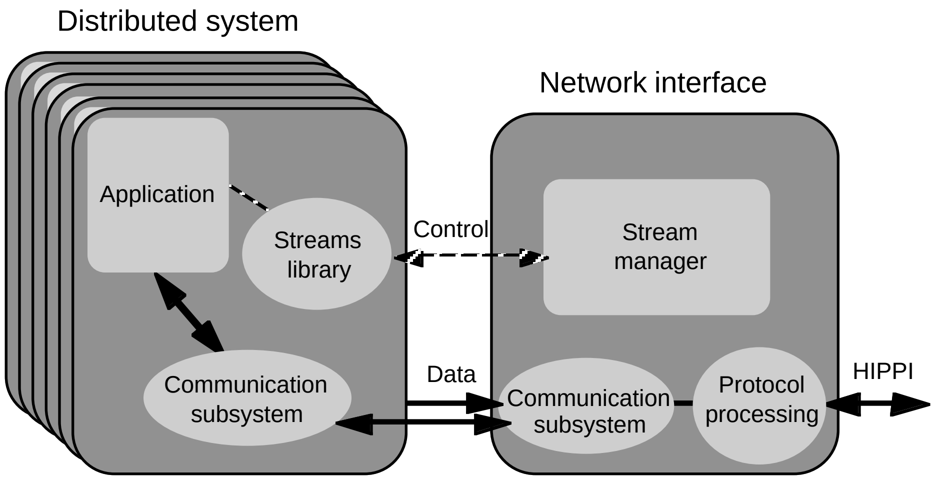


Figure 7: Streams architecture

In the streams architecture, the network interface implements a simple I/O interface, called the streams interface. The streams architecture is based on *logical connections*. Each logical connection supports communication between the application on a distributed-memory system to a destination on the external network. We define a *stream* as a data transfer over a previously established connection. For example, a stream could represent a series of frames to be displayed on a framebuffer.

To support streams while meeting the requirements listed above, we divide the responsibilities between the network interface and the distributed-memory system as follows (Figure 7):

- The *stream manager* on the network interface is responsible for efficient data movement of multiple streams from the interconnect of the distributed-memory to the network. It is also responsible for invoking the appropriate communication protocol, which formats headers and prepends them to each packet.

- The *application* on the distributed-memory system is responsible for distributing (or collecting) the data to (or from) the network interface node using existing communication libraries. This architecture does not imply that each application has to provide the code to transfer data to and from the network interface. For example, libraries can be built for common data distributions (e.g. [8]).

The components interact through a data interface and a control interface:

- The data interface transfers data between the network interface and the application on the distributed memory system. Two sets of parameters have to be agreed upon before a data transfer can take place: data format (quantity and ordering if striping is used), and local address information that will allow the distributed-memory system and network interface to exchange data.

- The control interface allows the application to instruct the network interface on how it should perform communication. Typical operations include opening or closing a connection, issuing an I/O operation, or inquiring about status. The control interface uses a flexible protocol that can easily be augmented to provide additional functionality for special devices or protocols (e.g. resetting the framebuffer, etc...).

Before communicating, applications first create a logical connection using the control interface. This request specifies the destination and protocol and returns a connection identifier. The application can then request the transfer of one or more data streams. Each data stream request specifies the connection identifier, data format information, local addressing information and (optionally) sequencing information relative to other streams. For transmit requests, the stream manager analyzes each request and translates the stream into a sequence of efficient data transfers from the internal interconnect into staging memory. There, data is formatted using the requested protocol and sent over the external network, as described in the previous section. Incoming data is stored on the network interface until a receive request is received from the application. At that point, the stream manager issues a sequence of transfers from network memory to staging memory, from where the data is DMAed onto the internal interconnect of the distributed-memory system following the data format instructions provided by the application.

This architecture meets the requirements we listed in the beginning of this section. By combining the streams package with data reshuffling (Section 6) a variety of data distributions can be supported efficiently. Applications can specify the required protocol at connection set up, but the data interface is protocol and device independent. Finally, the control interface supports striping and the timing of I/O operations under application control: striping is controlled through the data format parameters (Section 5.2) and timing is controlled by placing explicit constraints on the order of individual data transfers (Section 5.3.2). These features allow applications to create efficient, customized I/O operations. One important scenario is that of applications that have been parallelized using higher-level programming tools such as parallelizing compilers. Such tools already manage the resources in the distributed-memory system based on an understanding of the characteristics of the application, and they can also generate optimized I/O operations by selecting the right data movement libraries and inserting calls to the stream manager.

The alternative to the streams architecture is to have a more monolytic I/O package that is built into the operating system or supported by a general library. While this approach will be adequate for many applications, it will be inefficient for applications with very demanding or special I/O requirements. In Section 7 we present a number of examples that illustrate the importance of being able to customize the implementation of I/O operations.

5.2 Data Format Specification

Network I/O performance depends critically on how efficiently data is moved between the network interface and the distributed-memory system. This data transfer typically takes place across multiple links to the network interface and multiple connections have to be managed at the same time. In the streams architecture, the application provides the streams manager with information about the data format in advance, so that the stream manager can optimize and plan the data transfers, as is described in Section 5.3. The data format information specified by the application consists of the *Application Data Unit*, the *Transfer Unit* and the *Striping Unit*. We explain these terms below.

The *Application Data Unit* (ADU) is the block of data sent or received in a single communication operation. It is typically a data set generated or consumed by all compute nodes in a single iteration of the application. In this model the applications alternate between computation and communication phases, and during the communication phase they send or receive one or more data structures, each forming an ADU. Examples of ADUs include a single frame destined for a frame buffer or new values for a data structure (e.g. matrix) that are periodically received from another supercomputer. For most applications, all ADUs in a single stream will be identical. The stream model also supports variable sized ADUs, as we describe later.

The *Transfer Unit* (TU) specifies how ADUs are exchanged between the distributed-memory system and the network interface. A TU is a block of contiguous data that will be sent as a unit, possibly across multiple links. This unit is required since the amount of data that can be handled at any given time in a single unit of operation is limited by the available resources on the network interface node.

The *Striping Unit* (SU) is the amount of data within one TU that is transferred across a single link. The SU is derived from the TU and the number of links. The order of the SUs in a TU and the number of links

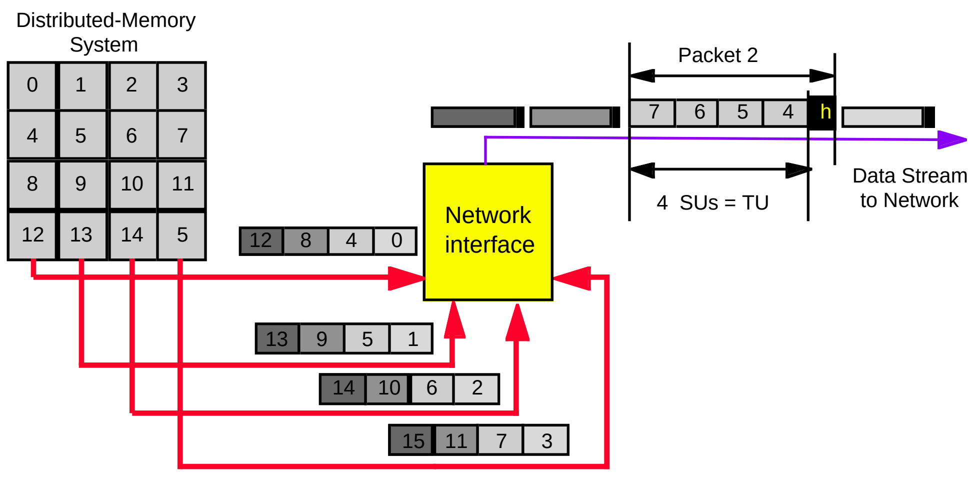


Figure 8: Data formating example

must be specified by the application.

A final parameter needed to completely specify the data format of a stream is the packet size on the external network. For some protocols, e.g. TCP/IP, the packet size is selected by the protocol. In other cases, e.g. raw HIPPI or UDP/IP, the application has to select the packet size. Typically a packet will map nicely onto one TU.

Figure 8 shows a simple example of how applications format data streams. In this example, the applications is sending a 128 KByte data that is distributed as 16 blocks of contiguous data, each if which is mapped on a processors of a 4x4 system as shown on the left in Figure 8. One way of implementing the data transfer is to have the nodes in each column of the distributed-memory system send their data over a separate link, as is shown in the center of the figure. The network interface will combine four SUs, one from each link, into a TU and add a header to form a packet that is sent over the HIPPI network. The data format parameters for this data stream are an ADU of 128 KByte, a TU of 64 KByte, and a SU of 8 KByte. Four packets are sent.

While many applications have regular data structures and all ADUs will have the same known size, some applications are not that regular, e.g. the ADU size can be data dependent. The streams model deals with these applications by having the ADU be the TU. While these streams require more involvement from the stream manager (i.e. reducing the granularity of resource control), they still supports efficient communication and they increase the class of applications that can use the *streams software*.

We will present some more detailed examples when we discuss the applications in Section 7.

5.3 The iWarp Streams Software

We give a more detailed description of the streams implementation for iWarp.

5.3.1 Data and control interface

The data interface between the distributed-memory system and the network interface is based on the PCS and ESPL communication libraries [24, 7]. PCS is used to create application-specific connections, and ESPL is a fast spooling library that achieve bandwidths close to the 40 MByte/second link rate, even for short messages. To support striping, we developed the Enhanced Multiple SPool Library (EMSPL), an extension of ESPL that manages communication over multiple links in an integrated fashion, thus minimizing global communication overhead.

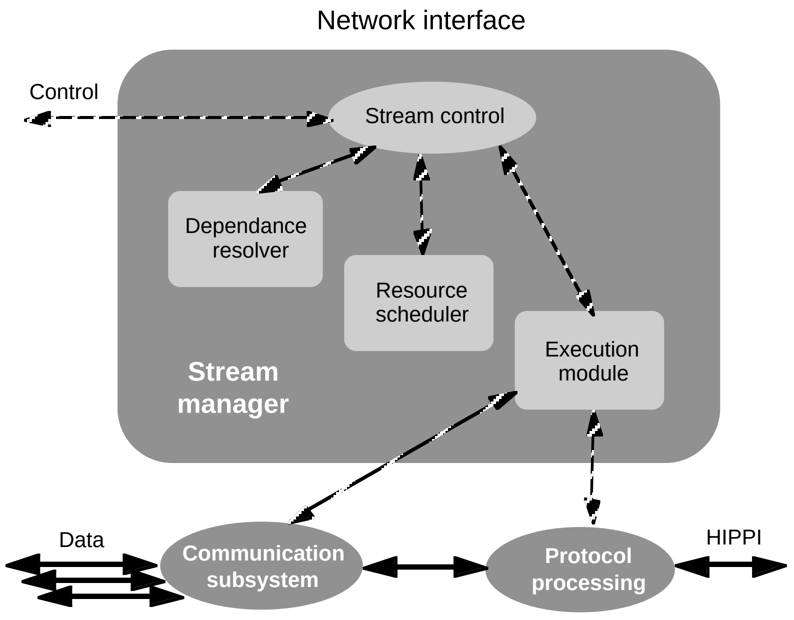


Figure 9: Structure of the stream manager

The control interface between the distributed-memory system and the network interface consists of a remote procedure call package that was implemented on top of the iWarp Message communication library (*imsg*). This library uses a communication backbone managed by the iWarp runtime system. The *imsg* library does not provide high-throughput communication, but it allows any pair of nodes to communicate without using additional system resources and without requiring a communication setup phase. For these reasons, it is appropriate for the exchange of control information. The imsg library also runs on the host that manages the iWarp system, enabling external programs to use the same control interface as the application, to manage the HIPPI interface if needed.

5.3.2 Stream Manager

Figure 9 shows the structure of the stream manager on the HIPPI interface of the iWarp system. It has three major components: the *dependence resolver* schedules ADUs subject to constraints specified by the application, the *resource scheduler* determines whether a stream can be scheduled, based on the available resources and the *execution module* is responsible for optimized data transfers. The *stream control* unit orchestrates the above modules. It interacts with the application through the control interface and schedules the other modules. iWarp does not support threads, so modules are scheduled explicitly.

Every stream that was registered by an application on the distributed-memory system has an associated state that specifies which module is handling it. Initially an *inactive stream* is handled by the dependence resolver. After all of its dependencies have been resolved it becomes a *pending stream* and it is managed by the resource scheduler. At any given time, multiple streams can be pending, all competing for the network node's resources. The resource scheduler is responsible for allocating resources to pending streams, by swapping them in and out of the execution module, i.e. making them *active streams*.

It is often important for applications to be able to specify that transfers take place in a certain order. Streams using the same logical connection are sequentialized by the dependence resolver according to the order they were registered with the stream control unit. Streams from different sources (on receive) may need to be handled by the application in a specific order. This is made possible through *stream sequencing*.

Also, at a finer granularity, an application may want to ensure that a group of different streams (usually to the same destination) progress in lock step. This is made possible through *stream synchronization* at ADU intervals. This feature can be used to synchronize multiple images displayed on the same framebuffer. Stream sequencing and synchronization are implemented by the dependence resolver, which orders data transfers in a manner specified by the application. In addition, the dependence resolver is used when scheduling a stream the first time to ensure that the connection to the destination exists and that there is enough data available.

The resource scheduler schedules pending streams based on the availability of the network interface's resources. For the iWarp system, the resources consist of buffer space in staging memory and DMA engines. Staging memory is a dual-ported RAM where data is briefly "staged" before it is placed in a larger network memory (on transmit) or sent to the distributed memory system (on receive). That memory is small because of the high bandwidth and random access requirements (due to striping). The iWarp chip only has a limited number of DMA engines (total of 6) to transfer data between the network interface and the distributed memory system. While these resources are clearly specific to iWarp, we expect other systems to have similar resource limits.

Resource management is done in a coarse grain fashion using the Application Data Units (as opposed to Transfers Units, for example) to minimize overhead. Applications can specify the number of ADUs a stream has to progress (in the execution module) before being swapped out, effectively allowing them to prioritize *pending streams*. An application can also specify that resources should be associated with a stream permanently, thus giving that stream the highest priority. Interleaving multiple streams at ADU boundaries is particularly suitable because it allows the application to pipeline the sending or receiving of data with reshuffling for example. This coarse grain management is possible because distributed memory systems typically operate on large blocks of data. It is used both for managing streams inside an application and from multiple applications that are time- or space-sharing the system.

The execution module is optimized to efficiently move data between the network and the distributed memory system. It operates on a limited number of active streams, each with its own set of pre-allocated resources. For each stream, the execution module uses double buffering. For example on transmit, the execution module first starts up the data transfer using spools (EMSPL – see above), and then invokes the communication protocol, which creates the header while the data is flowing into the buffer. The execution module then waits for the data transfers to finish, starts up the transfer of the newly filled buffer to the network (DMA), and switches buffers. This is repeated for each TU in the ADU. Double buffering and overlapping protocol processing with the data transfers achieves high throughput even if only one stream is active, which is the common case. Each of the buffers is large enough to hold both data and a header, allowing protocols to generate the packet header "in place". Several protocols are supported including raw HIPPI, IP and support for the NSC and IOSC framebuffers.

The execution module is most efficient when the ADU is an integral number of TUs. In this case it does not have to deal with "irregular" blocks. Since most applications distribute data structures in a regular fashion, the most natural ADU and TU will typically have this property. Buffer management and header creation are also simplified when a packet holds exactly one TU.

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 links to achieve higher throughput. For example, Figure 10 shows the bandwidth into an iWarp node

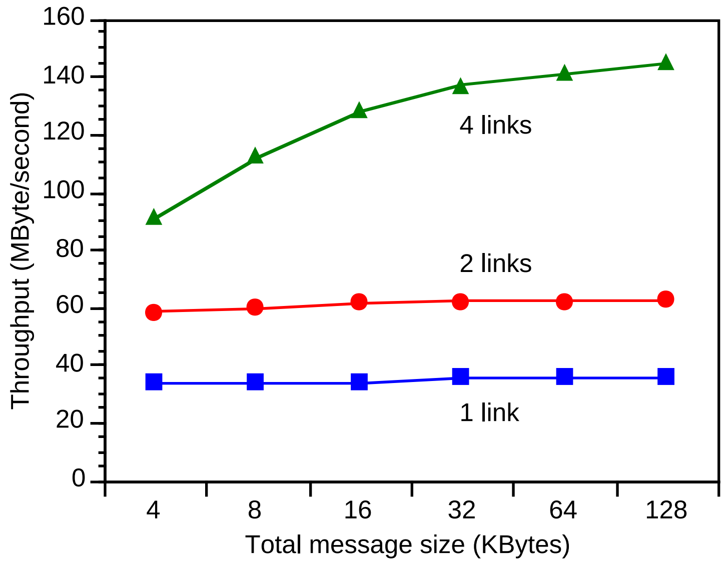


Figure 10: Increasing throughput using striping on iWarp

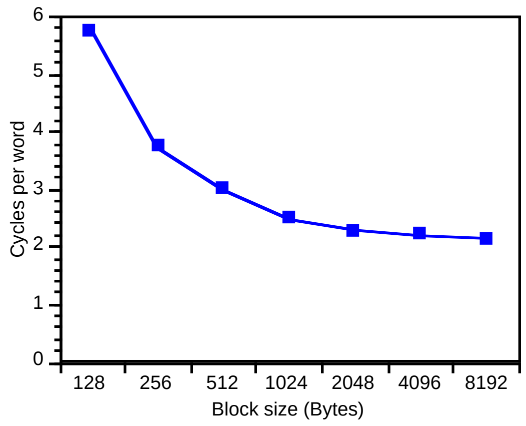


Figure 11: Cost of data movement

as a function of the message size and the number of links used for the transfer. We see that for all message sizes, the throughput using a single link can be almost doubled or quadrupled by striping the message over two or four links.

In this section we first describe our approach to data reshuffling and we then use the commonly used class of block-cyclic distributions to illustrate our approach.

6.1 Approach

A first solution is to have the network interface deal with the problem of data distribution: nodes send their data to the interface independently and the data is sorted and grouped by the network interface. While this approach is simple, it has the disadvantage that the cost of handling many small blocks of data can easily create a significant bottleneck on the network interface. Figure 11 shows for example the overhead per (4 byte) word on iWarp for sending blocks of different sizes over the internal interconnect, using spooling operations [7]. The overhead is calculated by dividing the number of cycles needed for the transfer by the number of words in the transfer. We see that the per-word overhead is much higher for small messages than for large ones and handling a lot of small blocks on the network interface will introduce a high overhead. Moreover, the network interface has to perform other tasks associated with communication (e.g., managing the network device, protocol processing) and the high communication overhead will consume cycles that are needed for these tasks. This will limit the sustained communication throughput. On systems where communication over the internal interconnect is less efficient than on iWarp, this effect will be even more significant.

In the approach taken, the distributed-memory system is responsible for the scatter/gather operation. On transmit, it constructs large messages and presents them to the network 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 node only has to deal with large blocks of data, independent from the data partitioning inside the system. 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 distributed-memory system can reshuffle data efficiently. A similar approach has been proposed for disk I/O [5].

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 an *I/O distribution*. The I/O distribution is a mapping from which data can be sent efficiently to the network interface. Then, using this favorable data distribution, the data can be sent to the network interface efficiently.

6.2 Example: block-cyclic distributions

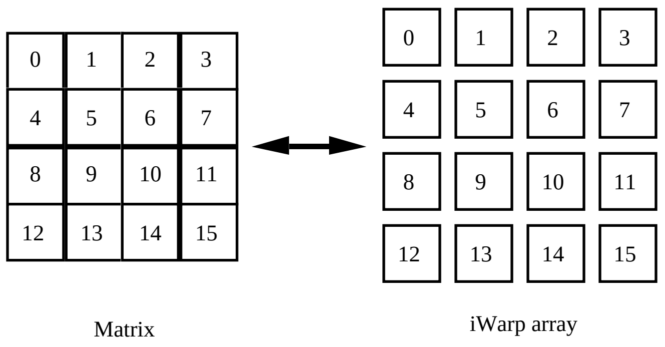
We illustrate our approach by describing a library that supports reshuffling for of block-cyclic data distributions.

6.2.1 Common data distributions

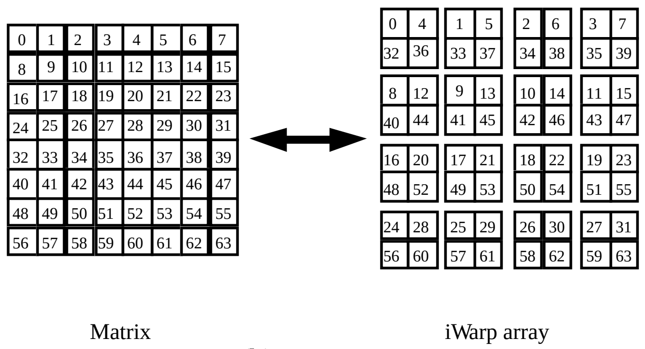
Applications on distributed-memory systems have very diverse data distributions. Figure 12 shows some of the data mappings that are used by iWarp applications. The row-swath partitioning is used by the Adapt image processing environment [46]. The coarse-grain block partitioning is used by several image processing applications and fine-grain block partitioning is used in the iWarp implementation of the LAPACK library [30]. These three examples are instances of block-cyclic partitionings: the data set is divided in blocks, which are distributed in a cyclic fashion across either the rows, or the rows and columns of the distributed-memory system. Block-cyclic distributions are widely used by applications and are for example supported by languages such as High-Performance Fortran [17].

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 12 given a $N * N$ matrix mapped on a $p * p$ processor system, 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 1000x1000 matrix distributed on an 8x8 system. A row-swath distribution would have 64 blocks of 16000 elements, a coarse-grain distribution would have 8000 blocks of 128 elements each, while a fine-grain block mapping would have 16000 blocks of 64 elements ($b = 2$).

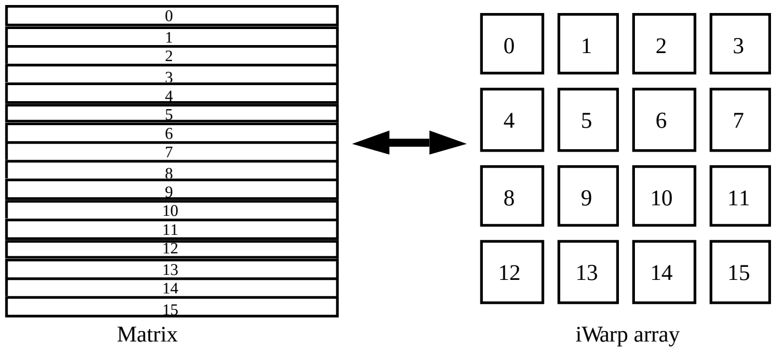
In the remainder of this section we describe a library that performs reshuffling for data mapped according to a block-cyclic distribution. Note that parallelizing compilers that map a program onto a distributed-memory system have enough information to insert the appropriate calls to do reshuffling in preparation for



(a) Coarse-grain



(b) Fine-grain



(c) Row-swath

Figure 12: Common data distributions

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

I/O.

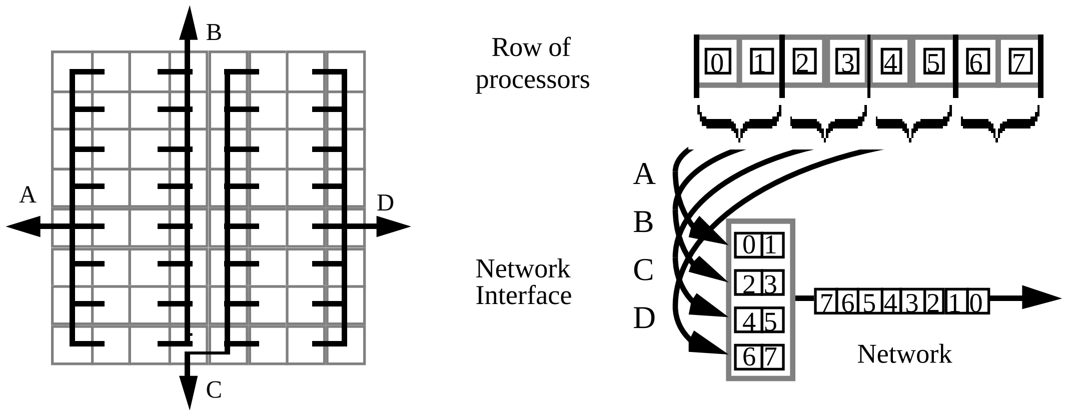


Figure 13: Data paths (left) and data flow (right) in the iWarp system used for I/O starting from the I/O mapping

6.2.2 I/O mapping

The first step in implementing the scatter/gather operation for block-cyclic distributions is to define an appropriate I/O mapping. Both the transmit and receive halves of the HIBs are connected to the distributed-memory system through four links. An attractive way of streaming the data from the system to the HIB is to have each column pair use one of the links, as is shown in Figure 13(left). This mapping has the advantage that the four data streams to the HIB use different links.

Figure 13(right) shows how data from the 8 nodes in a row of the iWarp system can be merged into a contiguous block of data on the HIB. Each node in the row holds one eighth of the data. The four pairs of nodes combine their data (in parallel) and forward the data to the HIB over a separate link. This is an efficient way for the HIB to receive data since it maximizes block sizes. Note that the amount of high-speed memory on the HIB is relatively small, so 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. The I/O mapping chosen corresponds to a block cyclic distribution. The algorithm used to reshuffle data from an arbitrary block-cyclic distribution to the I/O distribution is described in [8].

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

6.3 Performance measurements

We evaluate the impact of reshuffling using a 0.5 MByte data structure that is stored on an 8x8 iWarp system following block-cyclic distributions with different granularities. The specific distribution used has a block distribution y dimension, so each of the eight processors in a row of the iWarp system jointly hold 64KByte of contiguous data, and a block cyclic-distribution with block sizes ranging from 128 to 8192 bytes in the x dimension.

Figure 14 shows the rate at which we can perform data reshuffling using two different reshuffling algorithms. With 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 iWarp communication agent on each node is responsible for the forwarding of data going through a node, thus reducing the load on the processor and improving performance. We observe that both algorithms easily support throughputs well in excess of the target HIPPI rates. The reason is that the reshuffling algorithms are highly parallel: they exploit both parallelism between rows in the iWarp system and inside each row of iWarp [8].

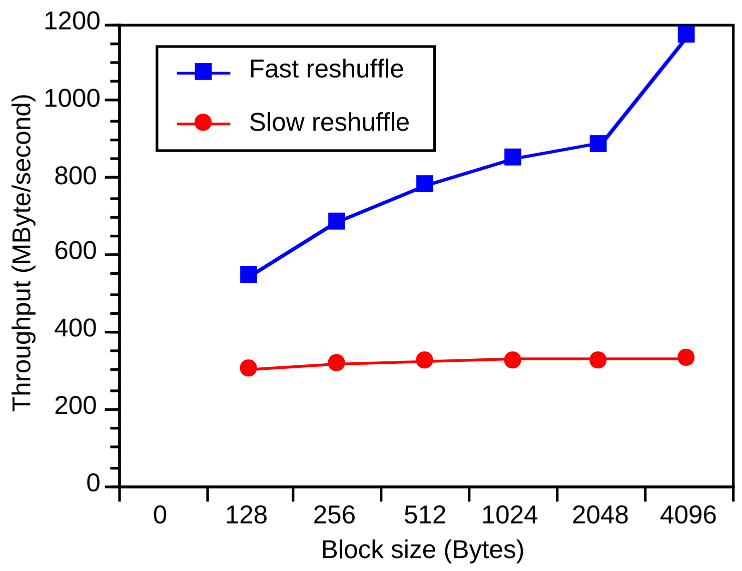


Figure 14: Performance of the reshuffling algorithm

Figure 15 shows the throughput with which we can transfer data to the HIB, starting with data mapped on the system using distributions with different block sizes. The figure shows four results. The curve labeled "streaming" is the throughput that can be achieved if each node sends its data directly to the HIB where the data is stored disregarding correct data ordering; this represents the upperlimit on the throughput we can expect for a functionally correct data transfer. The graph labeled "No Reshuffling" corresponds to the case where each node forwards its data to the HIB, where the blocks are stored in the correct order. We observe that for small block sizes the throughput is well below the HIPPI rate; the reason is that the HIB cannot receive and order the small data blocks fast enough. The graphs labeled "Slow Reshuffling" and "Fast Reshuffling" first reshuffle the data using one of the two algorithms described above, and then send the resulting large data blocks to the HIB. This approach allows us to achieve our target HIPPI rates, for both algorithms. Note that we come close to the maximum rate, represented by the "Streaming" curve, and that in the case of 8KByte blocks all approaches give the same performance because no reshuffling is needed.

The results with reshuffling in Figure 15 exhibit an interesting anomaly: the throughput using the slow reshuffling algorithms is consistently slightly higher than the throughput using the fast algorithm. This behavior is caused by contention for resources between the reshuffling algorithm and the transfer to the HIB. These two communication operations overlap and share some links in the system. Since the transfer to the HIB is the slower of the two and limits overall overall throughput, one would like it to have priority over the less critical reshuffling. However, in iWarp, multiple connections sharing a link share the link bandwidth evenly, and as a result, the reshuffling slows down the transfer to the HIB. The slow down is more significant with the fast reshuffling algorithm, since it uses more bandwidth. This anomaly could be avoided on an interconnect that supports the reservation of bandwidth for critical connections, as is for example being explored in the networking community [9, 3].

Figure 16 gives the breakdown of the time spent during the transfer to the HIB. The times are for a node in the last row of the iWarp system. The iWarp node performs the reshuffling and then waits for its turn to send data to the HIB. Multiplying the time spent sending data to the HIB by eight (to account for the eight rows of nodes sending their data) accounts for over 90% of the elapsed time, indicating that the reshuffling phase in one node often overlaps with the dispatching phase in others, i.e. reshuffling and the transfer to the HIB partially overlap. This explains why the throughput with reshuffling is close to the "Streaming" throughput for a raw transfer to the HIB.

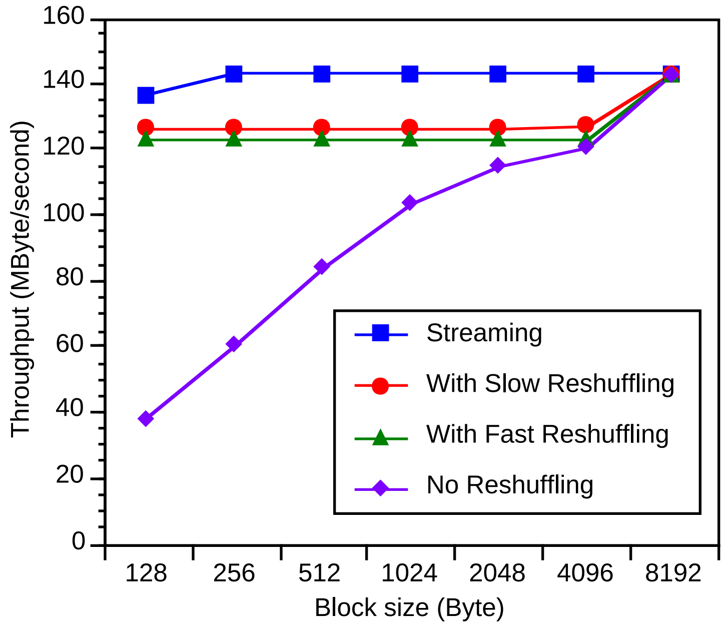


Figure 15: Throughput to the HIB

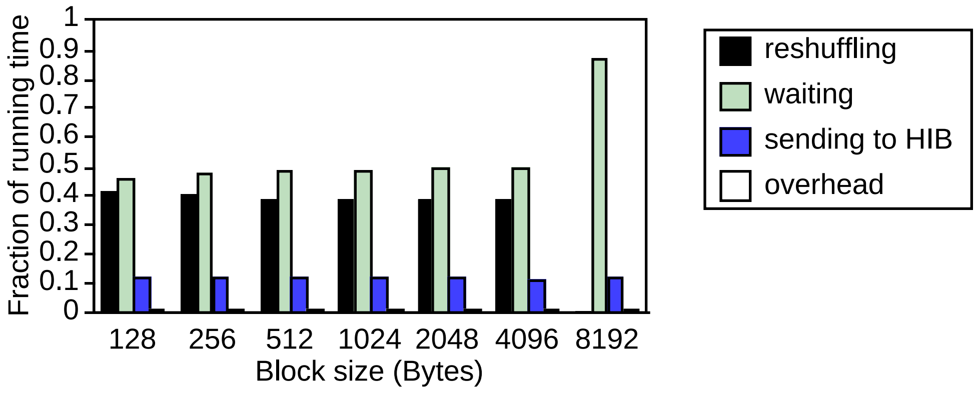


Figure 16: Breakdown of the algorithm execution time

7 Applications

In this section we describe how three very different applications use the iWarp HIPPI interface to communicate. We discuss both the communication performance and how the applications use the HIPPI interface. The diversity in the applications demonstrates the flexibility of the I/O architecture, specifically the streams software and application-specific data formatting.

In all the applications we refer to the network interface node of the iWarp system as the HIB (HIPPI Interface Board). All the iWarp systems were configured with 64 nodes. Some applications use an iWarp system that has 32 *large-memory nodes*, each with 2 MByte of memory. The rest of the nodes have 512 KByte of memory.

7.1 Chemical Process Optimization

The Chemical Process Optimization application was implemented by Robert Clay and Greg McRae of the Chemical Engineering Department at CMU. The application optimizes a system, for example a chemical

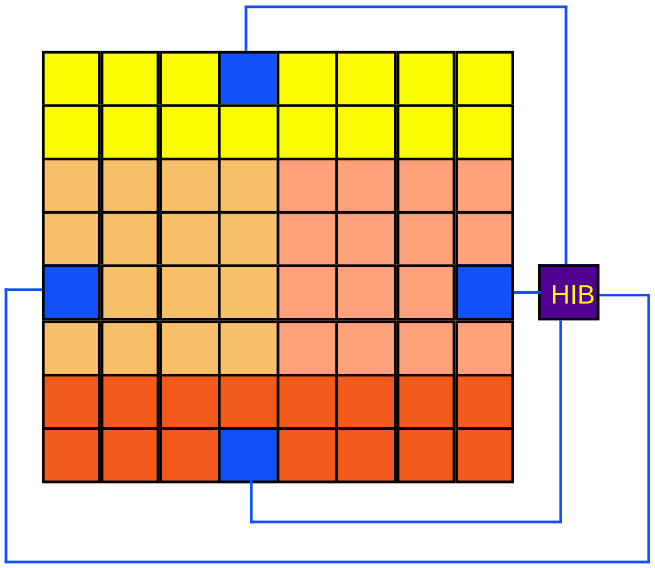


Figure 17: I/O configuration in the Chemical Process Optimization Application

plant, using a stochastic model [12] of the system. The stochastic optimization problem can be transformed into a deterministic problem using the certainty equivalence transformation [15].

Our implementation of the stochastic optimization problem [13] was distributed across a heterogeneous system: the Intel iWarp at the CMU campus and the Cray C-90 and TM CM2 at the Pittsburgh Supercomputer Center (PSC). Specifically, the iWarp is used to generate a complex probability manifold representing a plant. The generated data is then sent to the C-90 at PSC for analysis, and finally, the C90 and CM2 solve the resulting linear assignment problem using a heterogeneous solver [11].

We collected data for several application runs, corresponding to input sizes of 1k, 2k and 4k; the samples generated by iWarp in these runs correspond to 256 MB, 1 GB and 4 GB of data. The program divides the iWarp system into four quadrants, each consisting of 15 compute nodes that calculate data and one "forwarding node" that collects the results and communicates with the network interface node (Figure 17). The mapping of the application on iWarp was chosen in part to simplify I/O and as a result, there was no need to reorganize the data on the iWarp system. Each communication node forwards a SU of 16K byte, which is combined on the network interface to form a 64 KByte TU.

On a 64 node iWarp system, the application was able to generate 1400 Ksamples/second, maintaining a throughput of 6 MByte/sec to the C90. Measurements showed that the data generation was completely compute-bound: the four interface nodes where active only a fraction of the time and were operating in parallel with the data generation on the other nodes. In this application, the HIB and the streams software easily sustained the generated data bandwidth. A simple extrapolation showed that we would need a 512 node iWarp system before the communication becomes the bottleneck.

7.2 Medical Imaging

The goal of this project, which is headed by Dr. Doug Noll from the University of Pittsburgh Medical School, is to superimpose function and pathology information on a volume rendered brain. Basic brain researchers must accurately identify areas of brain involved with specific functions, while surgeons can utilize rendering to visualize approaches that spare areas of the brain. The current application represents the first step in this process: obtaining reconstructed, processed and rendered images based on Magnetic Resonance Imaging (MRI) data in a timely manner [34]. Our implementation is mapped on three architectures: the iWarp system

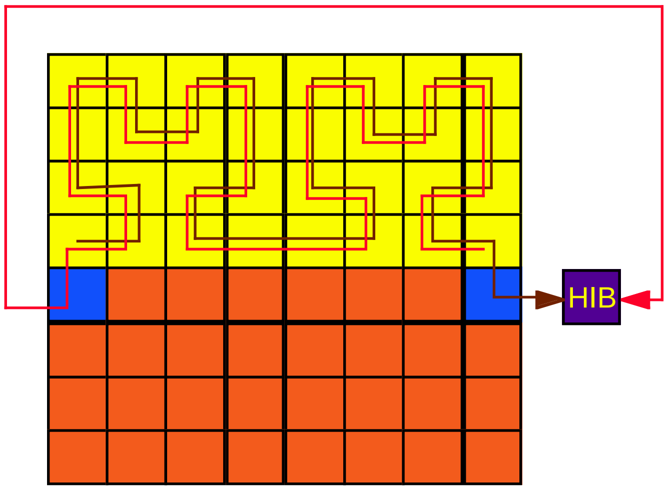


Figure 18: I/O configuration in the MRI Image Reconstruction application

at CMU performs pixel classification (brain/non-brain) and surface triangularization [35], the C-90 at PSC performs scalar processing, and the Intel Paragon at CMU performs the rendering.

The input to the iWarp component of the application consists of 52 MRI slices with pixel values representing the probability of being brain tissue. The data was block distributed on 32 large-memory processors of the iWarp system, each operating on 4-8 rows of image data. The iWarp generated approximately 580,000 triangles (22 MByte of data) that were rendered on the Paragon. Surface triangularization uses a modified marching cubes algorithm that removes poorly shaped triangles. Rendering uses a single light source, flat shading, z-buffered algorithm.

This application is particularly interesting from the streams interface point of view, because the amount of data to be sent out over the HIPPI network is data dependent, and there is no clearly definable Application Data Unit (ADU). This is due to the fact that while we are starting with a symmetrical distribution of data, with each iteration of the images, we end up with a different number of triangles in each node.

The solution was to set up a specific data collection mechanism for this application (Figure 18). We set up two routes, one in each direction, passing through all the participating nodes in a way that minimizes internal latency. At any given time only one node has control over the routes. That node forwards the triangles it generated (if any) to the HIB. After it has exhausted its data, it sends a short notification to the next node, effectively delegating the control over the routes. The latter allows pipelining communication and computation: each node can generate data for the next iteration while the other nodes in the route finish forwarding their own data. A similar algorithm can be used with a more traditional communication interface.

The application sustains a bandwidth of 40 MByte/second to the Cray C-90 at PSC. We had to use two routes and stripe the data across two links to the network interface to achieve this. The critical features in the streams software are the support for irregular data sets and for application-specific data distributions.

7.3 Stereo-Vision

In the stereo-vision application developed at CMU by Jon Webb, multi-baseline video images from four cameras are correlated to generate a depth image [45]. As part of that application, a "digital VCR" was implemented to display the four images on a framebuffer. The requirements called for real-time display along with the additional computations in the stereo-vision application.

Figure 19 shows the connections between the application on the system and the network interface. These connections are application specific: 32 large memories are used to store two interleaved fields from the 4

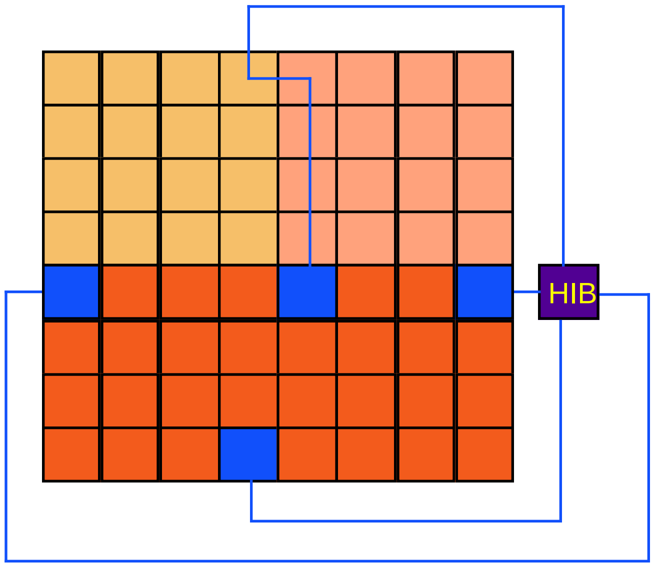


Figure 19: I/O configuration for digital VCR

cameras, while 26 other nodes reconstruct the 4 images and reshuffle the data. 4 interface nodes are used to forward the four images to the framebuffer through the network interface.

The HIB with the streams software was able display four black and white images at a resolution of 486x496 pixels at 30 frames per second, which corresponds to a sustained throughput of close to 30 MByte/second. The specific features of the streams software that were used include support for generating frame buffer headers using a "framebuffer protocol" and stream synchronization to keep the four images synchronized. Another experiment showed that the HIB with the stream interface can sustain 4 color images of the same resolution at rates of 12 frames per second or 46 MByte/seconds. Due to limited memory resources on the iWarp, this cannot be achieved on the production system.

8 Related work

A number of other research projects working on high-speed network I/O for distributed-memory systems have been described in the literature.

A group at Los Alamos National Labs has developed the CrossBar Interconnect (CBI) [43]). The CBI is an outboard protocol processor that performs protocol processing for supercomputers connected to HIPPI networks. It has two full-duplex HIPPI connections: one to the supercomputer and one to the HIPPI network. This design is driven by the observation that many distributed-memory systems cannot perform protocol processing at high rates, so it is attractive to develop a protocol processor that can be reused for different systems. The CBI and supercomputer exchange large blocks of data using the SHIP protocol, which provides support for addressing and flow control. The CBI performs protocol processing and sends/receives IP packets over the HIPPI network. The CBI has a large memory and checksum hardware, and is architecturally similar to the CAB. It is connected to a personal computer, which is responsible for control and protocol processing.

The main difference between the CBI and the HIB is that the CBI is a separate protocol processor that is attached to a network interface, while in the case of the HIB, protocol processing is performed on the network interface itself. While the CBI has the advantage that it can be reused for different systems, it adds a step to the I/O process, which will increase latency and reduce throughput for smaller data transfers.

The UNC HIPPI interface for the PixelPlanes system [39] is architecturally closer to the HIB, since protocol processing is performed on the network interface itself. However, by design it focuses on a specific application, while the I/O architecture described in this paper has been used for a variety of applications that use very different data and program structures on the distributed-memory system.

Several projects have looked at the difficulty introduced by the data distribution and representation. Kwan and Terstriep [31] 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. Data reorganization has been proposed to achieve high bandwidth access to disks in distributed-memory systems [5, 29, 38]. For example, in [5] the authors study the problem of implementing high-speed file I/O in the Intel Touchstone Delta. They 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 from ours. First, our target throughput is much higher. Second, we exploit the regularity of the distributions to achieve good reshuffling performance; this is important, given our performance goals. Note that the data reorganization is in effect a collective communication operation [18, 4].

9 Applicability to Other Systems

We have presented a description of the architecture and implementation of a high-bandwidth network interface for the iWarp system. In this section we examine how the results apply to other distributed memory systems.

9.1 Protocol processing

Our approach to protocol processing discussed in Section 4 is fairly independent from the iWarp architecture. While the details of our memory architecture are iWarp specific, the use of outboard buffer and checksumming are general technique that can be used to minimize the load on the memory bus, which is typically the bottleneck of high bandwidth host interfaces. Similarly, some of the features of our protocol implementation (e.g. separate transmit and receive processor) are specific to the HIB, but the optimizations to the API and buffer management are generally applicable.

9.2 The streams architecture

The streams architecture described in Section 5 readily applies to other distributed-memory systems. Specifically, the division of tasks between the distributed-memory system and the network interface, the separation of the control and data interface, and the ability to customize I/O operations all carry over unchanged. This is a result of the fact that the streams architecture is driven by the overall architecture of distributed-memory systems (e.g. balance of CPU and memory resources between network interface and the system) and application features (e.g. the use of different widely different data distributions and synchronization requirements).

The implementation of the streams architecture for other distributed-memory systems can potentially be quite different. We expect to see changes in the implementation of the control and data interfaces, the execution module and the control unit of the stream manager. The data interface will have to be reimplemented using the native communication interface for the system, e.g. Nx on the Paragon [25] or remote put/get on the Cray T3D [1]. Using a different communication interface will also affect the implementation of the execution module since it optimizes data transfers. Finally, the system software of most distributed-memory systems supports some form of multi-programming. This will simplify the implementation of the stream manager since the the scheduling of the modules on the network interface can be handled by the operating system or threads package, thus eliminating the need for the custom scheduler.

9.3 Data reshuffling

We use data reshuffling inside the distributed-memory system to create large blocks that can be handled efficiently by the network interface. This optimization is widely applicable and has for example also been used to optimize disk I/O on distributed-memory systems [5, 29, 38]. Systems that support very efficient communication, e.g. the remote memory reads and writes on the Cray T3D, may not require data reshuffling, but any system with non-negligible per-packet overhead is likely to benefit from data reshuffling for fine grain data distributions.

The iWarp HIPPI interface relies heavily on striping to achieve high sustained throughputs. Striping is essential because the internal link speed for iWarp is substantially lower than the HIPPI speed: 40 MByte/second versus 100 MByte/seconds. Several distributed-memory systems have higher internal link speeds, making striping less important for high-speed I/O on these systems. Note however that striping is a general technique for dealing with bandwidth a mismatch between the internal and external link speeds and since the speed of internal and external links is driven by technology, market and standardization factors that are largely independent, we should expect such mismatches to occur on a regular basis. As a result, support for striping should be built into the network interface architecture. Depending on the application bandwidth requirements, striping might be needed over multiple internal links, multiple external links, or both [20].

10 Conclusion

We presented an architecture for network I/O on distributed memory systems and described its implementation for an iWarp HIPPI interface. We were able to achieve fast I/O rates over the iWarp HIPPI interface for real-world applications. While these applications had different data distributions and different sets of requirements and constraints, they were all supported adequately and with good performance. The high throughput is made possible by three key features in the I/O architecture.

1. Application specific tasks are off-loaded from the network interface to the distributed memory system. This minimizes the load on the network interface, which is a bottleneck in many existing systems.
2. The streams manager on the network interface optimizes the network node's resource management and concentrates on efficient data transfers with a view of the whole distributed-memory system. Hardware support is used to optimize data touching tasks so memory bandwidth is used efficiently.
3. The tasks that are executed on the distributed-memory system (data management, communication with the network interface) are performed in close cooperation with the application so that optimization across the entire application, including I/O, is possible.

When the system is programmed using a programming tool, e.g. a parallelizing compiler, that tool is in an ideal position to manage the I/O, and our architecture provides the mechanisms to accomplish that task.

Acknowledgments

We thank all the individuals from the iWarp and Nectar groups at CMU who have contributed to this project. Of special mention are Michael Hemy, Todd Mummert, Brian Zill, Tom Warfel, Robert Clay, Chang-Hsin Chang, Doug Noll, Thomas Gross, Carl Love, Susan Shinrich, Claudson Bornstein, and the Network Systems team that built the HIB hardware. The paper also benefited from feedback from the anonymous reviewers.

We acknowledge the use of computational resources provided by the Pittsburgh Supercomputing Center (PSC).

References

[1] D. Adams. *Cray T3D system architecture overview*. Cray Research Inc., September 1993. Revision 1.C.

[2] ANSI. High-Performance Parallel Interface - Mechanical, Electrical and Signalling Protocol Specification HIPPI-PH. ANSI X3.183-1991, 1991.

[3] Jaime Jungok Bae and Tatsuya Suda. Survey of traffic control schemes and protocols in atm networks. *Proceedings of the IEEE*, 79(2):170–189, February 1991.

[4] M. Barnett, L. Shuler, R. van de Geijn, S. Gupta, D. G. Payne, and J. Watts. Interprocessor Collective Communication Library (InterCom). In *Proceedings of the IEEE Scalable High Performance Computing Conference*, pages 357–364. IEEE, May 1994.

[5] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel i/o. In *Proceedings of Supercomputing '93*, pages 452–461, Oregon, November 1993. ACM/IEEE.

[6] 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. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[7] 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. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 70–81, Seattle, May 1990. ACM/IEEE.

[8] Claudson Bornstein and Peter Steenkiste. Data reshuffling in support of fast I/O for distributed-memory machines. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, pages 227–235, San Francisco, August 1994. IEEE.

[9] Dave Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanisms. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 14–26, Baltimore, August 1992. ACM.

[10] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[11] R. L. Clay. *Scheduling in the Presence of Uncertainty: Probabilistic Solution of the Assignment Problem*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1991.

[12] R.L. Clay and G.J. McRae. Solution of Large-Scale Modeling and Optimization Problems Using Heterogeneous Supercomputing Systems. In *SuParCup 1991 proceedings*, Mannheim, Germany, 1991.

[13] Robert Clay and Peter Steenkiste. Distributing a chemical process optimization application over a gigabit network. In *Proceedings of Supercomputing '95*, page To appear. ACM/IEEE, December 1995.

[14] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network Magazine*, 7(4):36–43, July 1993.

[15] G.B. Dantzig. Planning Under Uncertainty Using Parallel Computing. Technical Report SOL 87-1, Department of Operations Research, Stanford University, 1987.

[16] M. de Prycker. *Asynchronous Transfer Mode*. Ellis Harwood, 1991.

[17] High Performance Fortran Forum. High performance fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, May 1993.

[18] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883, Oregon, November 1993. ACM/IEEE.

[19] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a high performance fortran framework. *IEEE Parallel & Distributed Technology*, 2(3):16–26, 1994.

[20] Thomas Gross and Peter Steenkiste. Architecture implications of high-speed I/O for distributed-memory computers. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 176–185, Manchester, England, July 1994. ACM.

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

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

[23] Anthony J. G. Hey. Supercomputing with Transputers - Past, Present and Future. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 479–489. ACM, June 1990.

[24] S. Hinrichs. *Programmed Communcation Service Tool Chain User's Guide*. Carnegie Mellon University, release 2.8 edition, 1991. Now part of Intel RTS 3.0.

[25] Intel Corporation. *Paragon X/PS Product Overview*, March 1991.

[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.

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

[28] Karl Kleinpaste, Peter Steenkiste, and Brian Zill. Software support for outboard buffering and checksumming. In *Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols*, pages 87–98. ACM, August/September 1995.

[29] D. Kotz. Multiprocessor file system interface. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201. ACM/IEEE, January 1993.

[30] H. T. Kung and Jaspal Subhlok. A new approach to automatic parallelization of blocked linear algebra computations. In *Proceedings of Supercomputing '91*, pages 122–129, Albequerque, November 1991. IEEE.

[31] Thomas T. Kwan and Jeffrey A. Terstriep. Experiments with a gigabit neuroscience application on the cm-2. In *Proceedings of Supercomputing '93*, pages 133–142, Oregon, November 1993. ACM/IEEE.

[32] Sigurd L. Lillevik. The Touchstone 30 Gigaflop DELTA Prototype. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 671–677. IEEE, April 1991.

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

[34] Doug C. Noll, J. C. Cohen, C. H. Meyer, and W. Schneider. Spiral k-space MRI of cortical activation. *Journal of Magnetic Resonance Imaging*, 5:49–56, 1995.

[35] Doug C. Noll, Jon A. Webb, and Tom E. Warfel. Parallel Fourier inversion by the scan-line method. *IEEE Transactions on Medical Imaging*, in press, 1995.

[36] David R. O'Hallaron. The Assign Parallel Program Generator. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 178–185. IEEE, April 1991.

[37] Erich Rutche and Matthas Kaiserswerth. TCP/IP on the Parallel Protocol Engine. In *Proceedings of the 4th IFIP Conference on High Performance Networks*, pages C2 1–16, Liege, Belgium, December 1992. IFIP, Elsevier.

[38] K. E. Seamons and M. Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Seventh International Working Conference on Scientific and Statistical Database Management*, pages 218–227. IEEE, September 1994.

[39] Raj K. Singh, Stephen G. Tell, Shaun J. Bharrat, David Becker, and Vernon L. Chi. A programmable hippi interface for a graphics supercomputer. In *Proceedings of Supercomputing '93*, pages 124–132, Oregon, November 1993. ACM/IEEE.

[40] Peter Steenkiste. Analyzing communication latency using the nectar communication processor. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 199–209, Baltimore, August 1992. ACM.

[41] Peter A. Steenkiste. A systematic approach to host interface design for high-speed networks. *IEEE Computer*, 26(3):47–57, March 1994.

[42] 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. In *Proceedings of the 4th IFIP Conference on High Performance Networks*, pages A3 1–16, Liege, Belgium, December 1992. IFIP, Elsevier.

[43] Richard Thompson. Los Alamos Multiple Crossbar Network Crossbar Interfaces. In *Workshop on the Architecture and Implementation of High Performance Communication Subsystems*. IEEE, February 1992.

[44] Lewis W. Tucker and George G. Robertson. Architecture and Applications of the Connection Machine. *IEEE Computer*, 21(8):26–38, August 1988.

[45] Jon Webb. Latency and Bandwidth Considerations in Parallel Robotics Image Processing. In *Proceedings of Supercomputing '93*, pages 230–239, Oregon, November 1993. ACM/IEEE.

[46] Jon A. Webb. Architecture-Independent Global Image Processing. In *10th International Conference on Pattern Recognition*, pages 623–628, Atlantic City, NJ, June 1990. IEEE.