Gigabit I/O for Distributed-Memory Machines: Architecture and Applications

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

January 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. We implemented an architecture for network I/O for the iWarp system with the following two key characteristics: first, application-specific tasks are off-loaded from the network interface to the distributed-memory system, and second, these tasks are performed in close cooperation with the application. The network interface has been used by several applications for over a year. In this paper we describe the network interface software that manages the communication between the iWarp distributed-memory system and the network interface, we validate the main features of our network interface architecture based on application experience, and we discuss how this architecture can be used by other distributed-memory systems.

1 Introduction

The most powerful supercomputers today are distributed-memory systems that link a large number of workstation class processors together using a high performance interconnect. Their success can be attributed to the fact that they are inherently scalable and provide relatively inexpensive computing cycles compared with traditional supercomputers. However, while distributed-memory systems are effective compute engines, network I/O has proven to be a problem. The reason is that network I/O is typically supported through an I/O node with the same computational power as one of the compute nodes, i.e. a workstation-class node has to support network I/O for a supercomputer.

The I/O architecture used for the HIPPI interface for the iWarp system [3], described in [19], relies on a careful distribution of the network functions (e.g. protocol processing, data formatting and connection management) between the distributed-memory system and the network interface to achieve high-bandwidth network I/O. We achieved sustained throughputs of 55 MBytes/sec for simple demo applications and 40 MBytes/sec in large heterogeneous computing applications. In our architecture, the functions in the upper layers of the protocol stack are performed by the distributed-memory system, thus reducing the load on the network interface and improving overall throughput. 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.

In the past year, the iWarp HIPPI interface has been used by several applications. Our experience shows that one of the more difficult functions in the I/O architecture is the management of the data flow between the distributed-memory system and the network interface. Since this function is performed jointly by the application running on the distributed-memory system and the runtime system on the network interface, it requires careful coordination. In this paper we discuss in detail how the data flow is managed by the *streams software*, and we use application examples to show how the I/O subsystem supports efficient network I/O under different conditions. The results clearly show the need to closely coordinate the management of the I/O data flow with the execution of the application.

The remainder of the paper is organized as follows. We first review our architecture for network I/O on distributed-memory systems (Section 2). We then describe the *streams software* (Section 3) and its implementation on the iWarp system (Section 4). In Section 5 we briefly describe a number of applications that use the iWarp HIPPI interface and we discuss how they use the features of the *streams software*. We outline the applicability of the streams architecture to other systems in Section 6, and we conclude in Section 7.

2 An Architecture for Network I/O

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. The role of the network interface is to forward data between the internal and external network. Figure 1 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. The sequential network interface tends to become a bottleneck; most of the performance problems can be attributed to the functions corresponding to the transport, network, session and presentation layers of the OSI network model (Figure 2).

In the iWarp HIPPI interface [19], these problems are addressed by mapping each task onto the subsystem that is the most appropriate for it (network interface or distributed memory system):

1. Communication protocol processing (transport and network layers) generally does not parallelize well. We map this task to the network interface, and we provide hardware support for time-critical

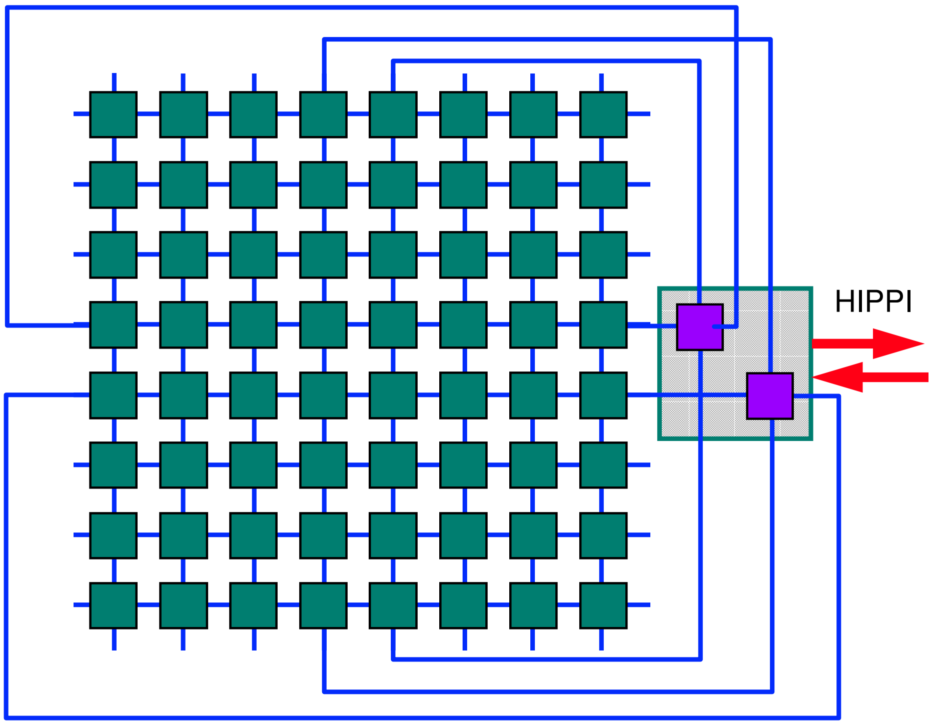

Figure 1: Connection of HIPPI network interface to iWarp distributed-memory system

tasks such as data checksumming and buffering [20].

2. Managing connections between the nodes in the distributed-memory system and the outside world through the network interface translates into a problem of allocating resources in the system (e.g. link bandwidth, buffer space, ..). We map this task to the *streams software*, as described in Section 3.

3. Data 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 or gather operations as part of the I/O operation [11]. The task of combining/distributing the data parallelizes well, hence we map it onto the distributed-memory system. This architecture alleviates the load on the network interface which can then handle the messages more efficiently.

Some of these techniques are similar to those used by other groups. For example, data reorganization has been proposed to achieve high bandwidth access to disks in distributed-memory systems [2, 14, 17] and the data reorganization is in effect a collective communication operations [10, 1]. Other systems also rely on the network interface to perform protocol processing. However, while we run a relatively standard TCP/IP implementation on the network interface, others use outboard protocol processing (e.g. [22]), or a customized protocol implementation relying heavily on hardware support (e.g. [18]).

This paper focuses on a key element of our architecture that has not been addressed systematically elsewhere: the management of connections and of data-flow between the distributed-memory system and the network interface. The architecture we have implemented using the *streams software* allows efficient use of resources by applications with different I/O loads, and efficient transfers for a variety of data distributions.

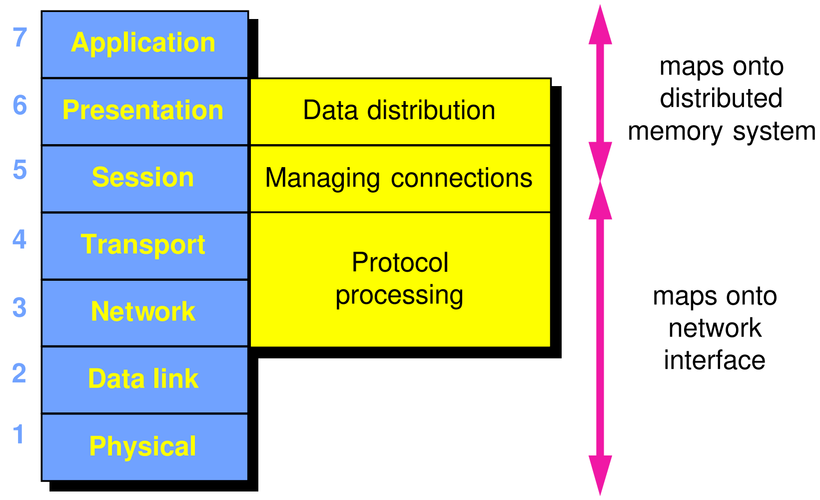

Figure 2: Mapping of protocol stack

3 The Streams Architecture

3.1 Requirements

Based on the above network interface architecture, we can derive the following requirements for the data movement between the application buffers in the distributed-memory system and the system buffers in the network interface:

- Support high bandwidth communication efficiently.
- Support different data distributions, including both regular and irregular distributions.
- Provide an application 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.

3.2 Architecture

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.

Handling streams with the requirements listed above, results in an I/O architecture that distinguishes between tasks that need to be performed on the distributed-memory system and tasks that need to be performed on the network interface. Specifically (Figure 3):

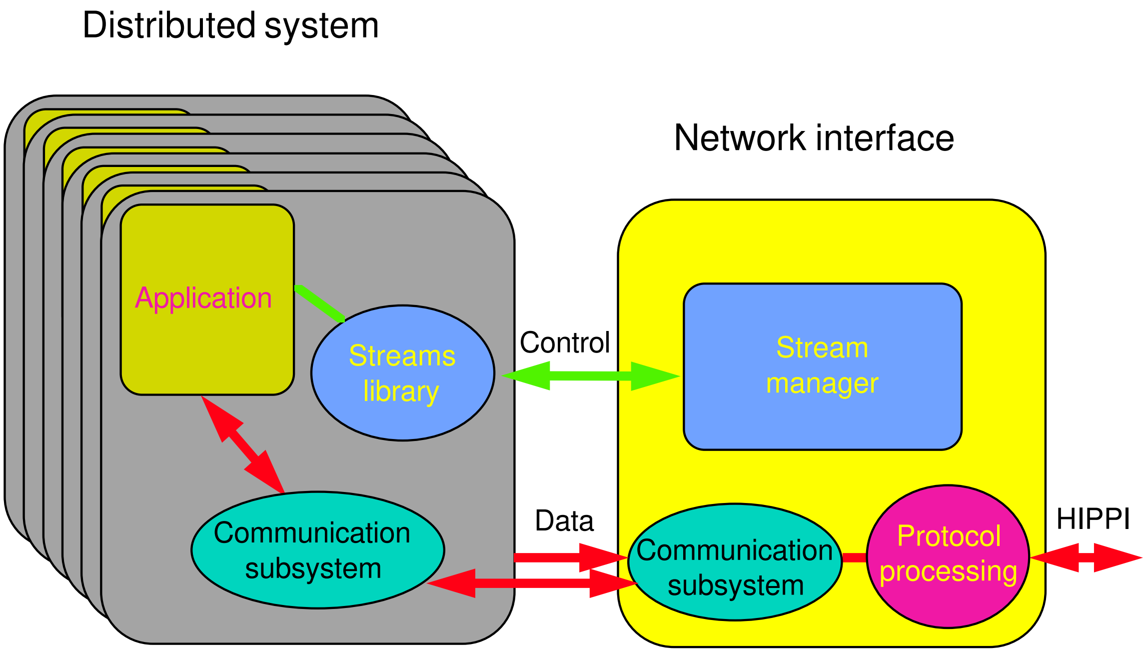

Figure 3: Streams architecture

- The **streams manager** on the network interface is responsible for efficient data movement from the interconnect of the distributed-memory to the network for multiple *streams*. 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. Rather, libraries can be built for common data distributions (e.g. [5]).

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

- The data interface transfers data between the 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:
    1. information on the data format (quantity and ordering if striping is used);
    2. 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.

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, specifying the connection identifier and the data format. For transmit requests, the stream manager analyzes each request and orchestrates efficient data transfers to the

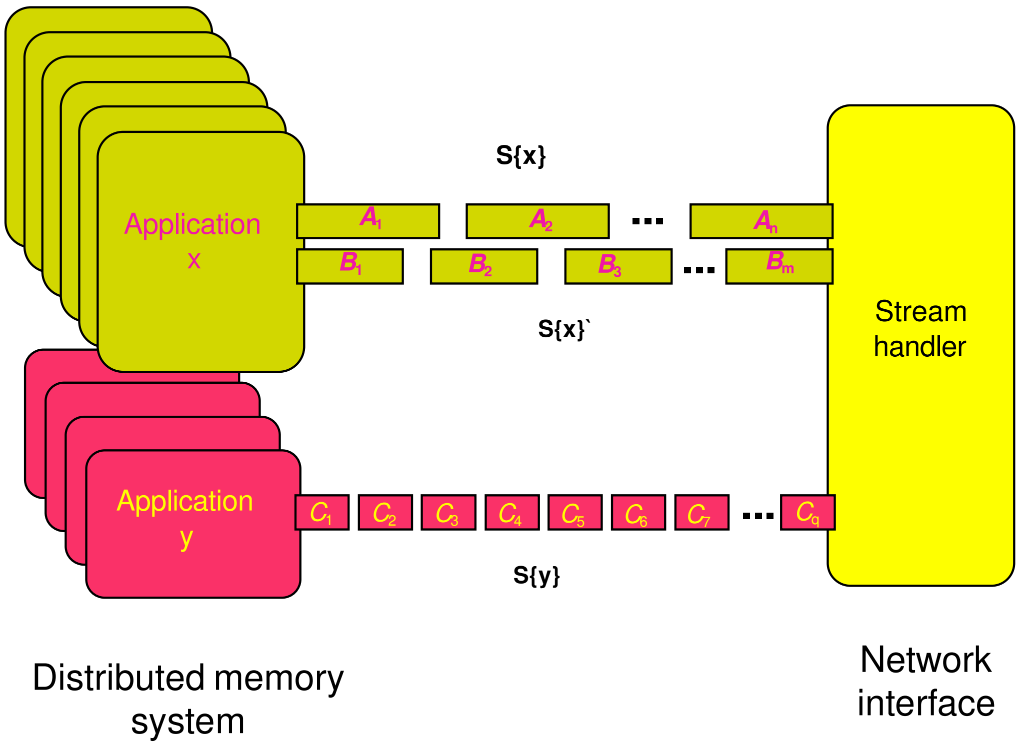

Figure 4: Space sharing applications using streams

network interface. There data is formatted using to the requested protocol and sent over the external network. For incoming data, the data is stored on the network interface until a receive request is received from the application. At that point, the stream manager transfers the data to the distributed-memory, following the data format instructions provided by the application.

An important aspect of the stream architecture is its ability to support I/O for streams from multiple applications that are time or space-sharing the system. This stems from the *streams software* being positioned at a high level where it can see the whole system and optimize I/O for the system as a whole. Figure 4 shows for example two applications, one with two open streams and the other with one, space-sharing the system.

If the distributed-memory systems is programmed using a programming tool, such as a parallelizing compiler, that tool is in an ideal position to manage the I/O, e.g. select the right data movement libraries and insert calls to the stream manager. The tool already manages the resources in the distributed-memory system based on an understanding of the characteristics of the application, and can optimize I/O operations internally (between the compute nodes) and externally (to the network interface node)

3.3 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 model, this management is performed using *Application Data Units*, *Transfer Units* and *Striping Units*, which are specified by the application. We explain these terms below, using Figure 5 as an example. It shows a data stream moving across 4 links between the distributed-memory system and the network interface node.

The **Application Data Unit** (ADU) is the block of data sent or received in a single communication

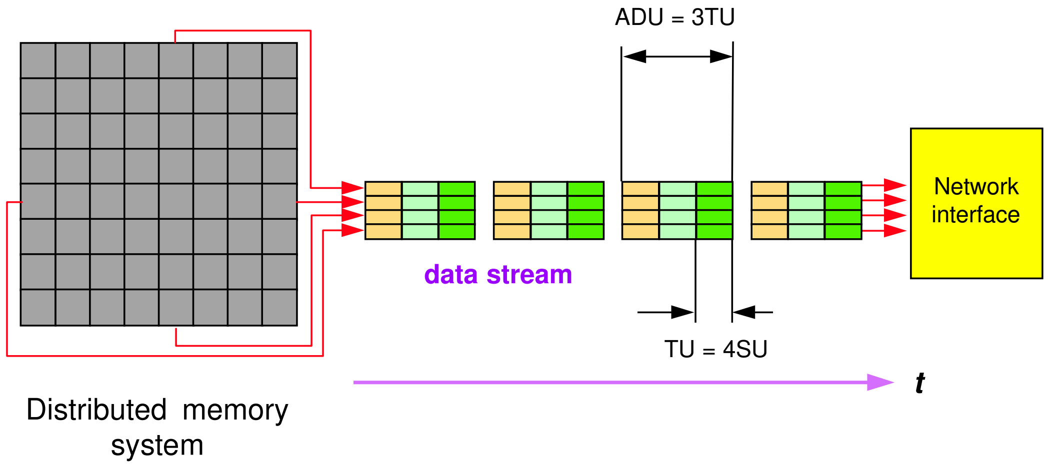

Figure 5: Stream data formatting example

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. A **stream** is a sequence of ADUs sent to, or received from, a specific destination on the external network. 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. Striping a TU across multiple links allows increasing the bandwidth (Figure 6). 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 must be specified by the application.

A final parameter needed to control 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 7 shows a typical example of how a data stream can be formatted for the network. The SUs are combined with a header to form a packet that is sent over the HIPPI network.

The **stream manager** uses the ADU and TU information to optimize data movement. Specifically, the TU is used to optimize buffer management, while the ADU information is used for coarse grain resource control. This is discussed in more detail in Section 4.3.2.

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 degenerating the ADU to be a TU. While these streams require more involvement from the stream manager (i.e. reducing the granularity of resource control), they still support 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 5.

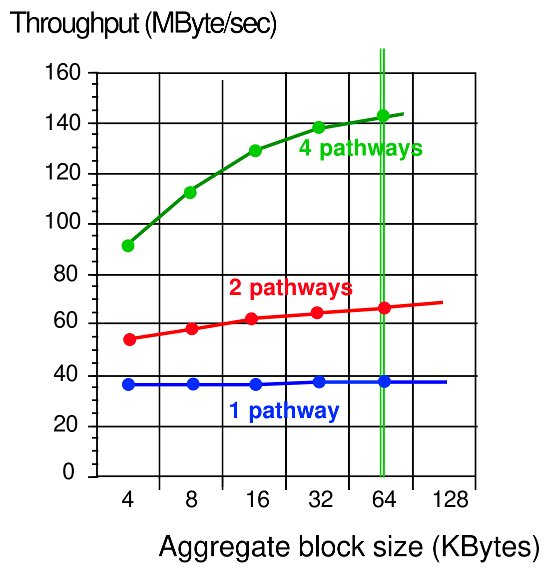

Figure 6: Increasing bandwidth using striping on iWarp

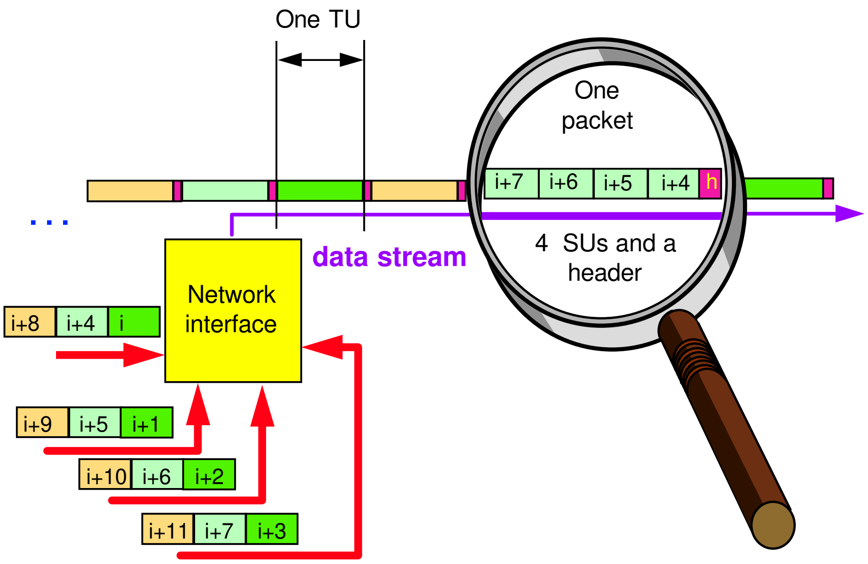

Figure 7: Formatted data on the network side

4 The iWarp Streams Software

In this section we give a more detailed description of the features that where implemented for the iWarp *streams software*.

4.1 Data Interface

The data interface between the distributed-memory system and the network interface is based on the PCS and ESPL communication libraries [12, 4]. PCS is used to create application-specific connections, and ESPL is a fast **spooling** library that achieves bandwidths close to the 40 MBytes/second link rate, even for short messages. Spooling is an iWarp mechanism for DMAing data directly to or from the node's memory. Spooling allows communication to take place in the background, freeing up the processor for other tasks.

The communication module on the HIPPI interface node needs to be able to collect/distribute data from multiple links of the interconnect simultaneously to support striping. This communication has to be performed in the background, so that other tasks, such as protocol processing, can execute at the same time. The required functionality is provided by the Enhanced Multiple SPool Library (EMSPL), an extension of ESPL that manages multiple links in an integrated fashion, thus minimizing global communication overhead.

4.2 Control Interface

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.

The control interface allows applications to create streams, query their status, and schedule them. It uses a flexible protocol that can easily be augmented to provide additional functionality. (e.g. resetting the framebuffer, etc...).

4.3 Stream Manager

Figure 8 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.
- The **execution module** is responsible for optimized data transfers.

The **stream control** unit orchestrates the above modules. It deals with the interaction with the application through the control interface and schedules the other modules. iWarp does not support threads, so modules are scheduled explicitly. The control unit interacts with the distributed-memory system only when the execution module is idling, minimizing the impact on the overall performance.

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 dependences 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 all pending streams, by swapping them in and out of the execution module, effectively making them **active streams**.

4.3.1 Dependence Resolver

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 (e.g. on receive) may

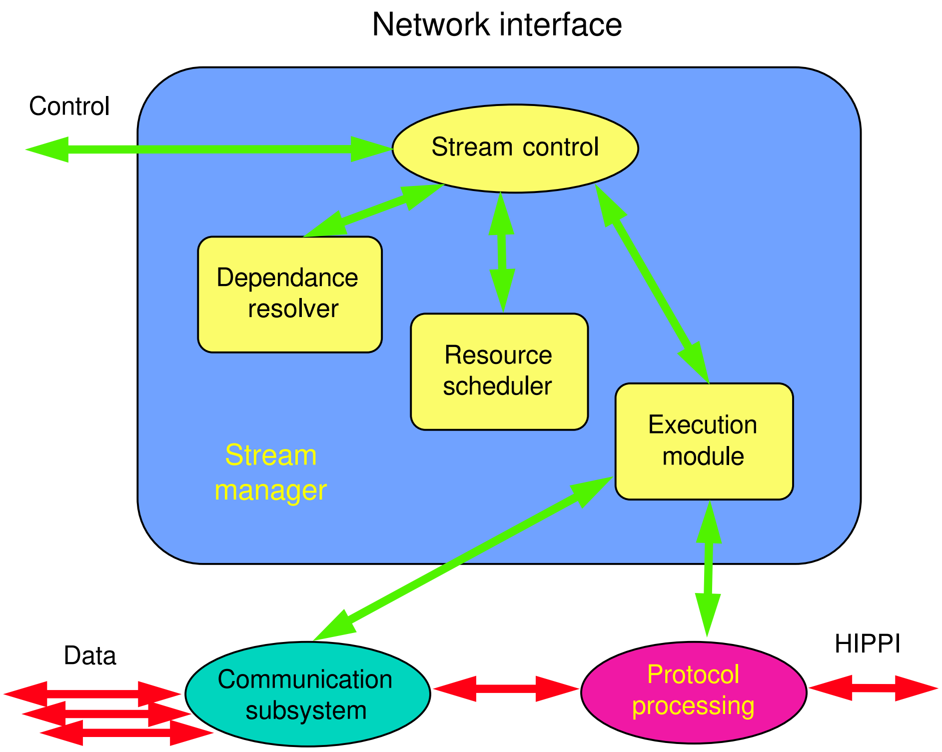

Figure 8: Stream manager's modules and structure

need to be handled by the application according to its computational stage. 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 data that needs to be handled.

4.3.2 Resource Scheduler

The resource scheduler schedules *pending streams* based on the availability of the network interface's resources. For the iWarp system, the following resources have to be available:

- buffer memory: the iWarp's network interface has a small, 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).
- communication resources: the iWarp hardware can support only a limited number of background spools (total of 6 - two are used by the runtime system) between the network interface and the distributed memory system. Spools are needed in order to handle multiple streams simultaneously or streams that require striping.

While these resources are clearly specific to iWarp, we expect other systems to have similar resource limits.

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.

The resource scheduler schedules the streams, by applying a best-fit algorithm designed to maximizes resource usage while progressing on all *pending streams*. Resource management is done in a coarse grain fashion using the Application Data Units (as opposed to Transfers Units for example.) to minimize overhead. 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.

4.3.3 Execution Module

The execution module is optimized to efficiently move data between the network and the distributed memory system. It operates on a limited number of streams, each with its own set of pre-allocated resources. The buffer memory can be partitioned (up to 4 segments) to enable processing multiple *active streams* at the same time. If the buffer memory is not partitioned, there can only be one *active stream*.

For each stream, the execution module uses double buffering. For example on transmit, one buffer is being filled with data coming from the distributed memory system, while the other is being formatted by the protocol processor and forwarded to the network. Double buffering (Figure 9) is possible because both the spools and the network node's DMA unit can work in the background. This mechanism 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 protocol used for a stream is specified by the application.

As can be seen in Figure 9, protocol processing is done in parallel with the data movement. The execution module first starts up the data transfer using spools (EMSPL – see Section 4.1), 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.

The execution module performs minimal management for each stream. This includes tracking the number of TUs processed per ADU, swapping streams with completed ADUs to the scheduler, detecting idle streams which hog resources and identifying *end of stream* messages. At some of these management intervals, depending on the state of the streams in progress, the execution module temporarily returns control to the stream scheduler. This scheme allows the execution module to be kept busy the highest percentage of time.

The performance of the execution module is optimal 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.
- A packet holds exactly one TU. In this case buffer management and header creation is significantly simplified.

4.4 Performance

Performance numbers for the basic operations have been reported elsewhere. Data can be moved between the iWarp compute nodes and the iWarp interface node at rates close to 145 MBytes/second. [19, 5]. The

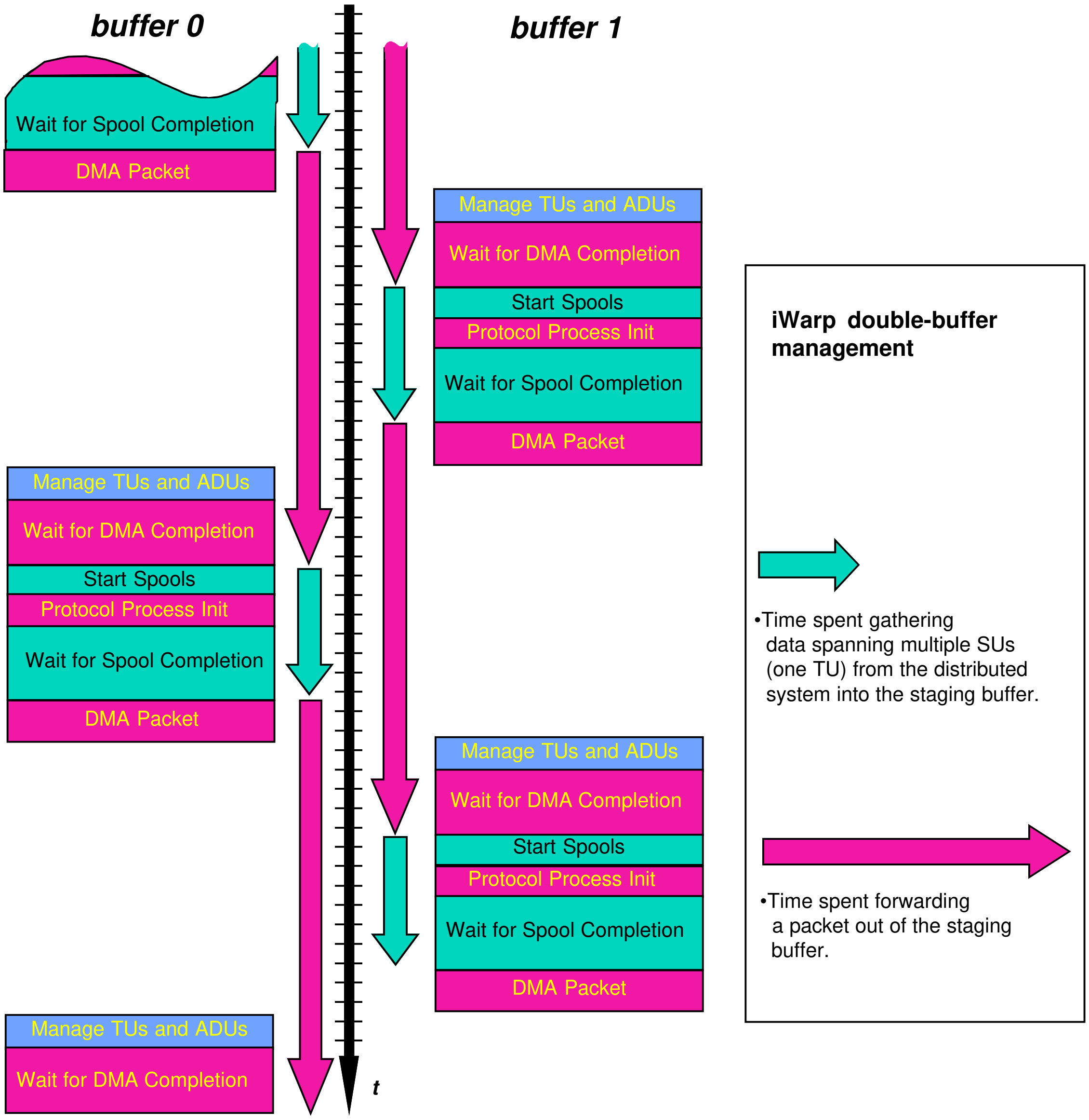

Figure 9: Double buffering in the execution module (transmit)

network interface can send data over the HIPPI network at 90 MBytes/second for 64 KByte packets [19]. Actual network I/O performance for applications is reported in the next section.

5 Applications

In this section we describe how four very different applications use the iWarp HIPPI interface to communicate. We discuss both the communication performance and how the applications use the *streams software*. The diversity in the applications demonstrates the flexibility of the *streams software*.

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 used an iWarp system that has 32 *large-memory nodes*, each with 16 MBytes of memory. The rest of the nodes have 512

KByte of memory.

5.1 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 plant, using a stochastic model [7] of the system. The stochastic optimization problem can be transformed into a deterministic problem using the certainty equivalence transformation [9].

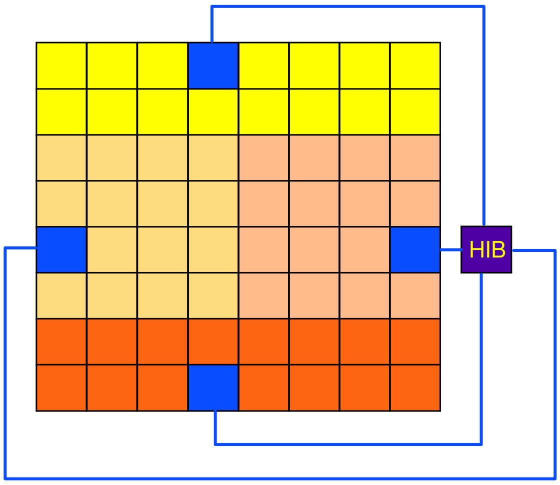

Figure 10: I/O configuration in the Chemical Process Optimization Application

Our implementation of the stochastic optimization problem [8] was distributed across a heterogeneous system comprising of: 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 [6].

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 10). The mapping of the application on iWarp was chosen to simplify I/O and as a result, there was no need to reorganize the data on the iWarp system. Each communication cell forwards a SU of 16K byte, which is combined on the network interface to form a 64 Kbyte TU.

On a 64 cell iWarp system, the application was able to generate 1400 Ksamples/second, maintaining a throughput of 6 MBytes/sec to the C90. Measurements showed that the data generation was completely compute-bound: the four interface cells where active only a fraction of the time and were operating in parallel with the data generation on the other cells. 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 cell iWarp system before the communication becomes the bottleneck.

5.2 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 [15]. Our implementation is mapped on three architectures: the iWarp system at CMU performs pixel classification (brain/non-brain) and surface triangularization [16], 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 MBytes 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 cell.

The solution was to set up a specific data collection mechanism for this application (Figure 11). We set up two routes, one in each direction, passing through all the participating cells in a way that minimizes internal latency. At any given time only one cell has control over the routes. That cell forwards the triangles it generated (if any) to the HIB. After it has exhausted its data, it sends a short notification to the next cell, effectively delegating the control over the routes. The latter allows pipelining communication and computation: each cell can generate data for the next iteration while the other cells in the route finish forwarding their own data. A similar algorithm can be used with a more traditional communication interface.

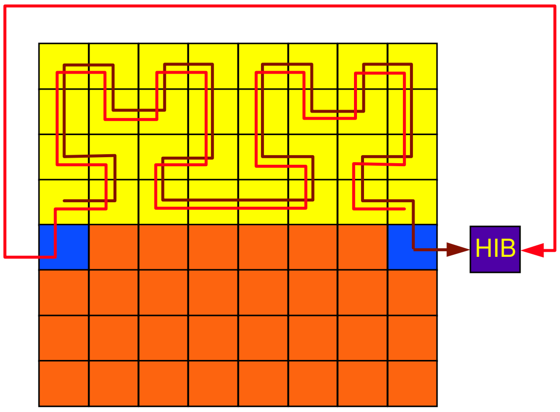

Figure 11: I/O configuration in the MRI Image Reconstruction application

The application sustains a bandwidth of 40 MBytes/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.

5.3 Block Cyclic Distributions

An earlier paper [5] describes how the creation of large contiguous data blocks using reshuffling in the distributed memory systems can be used to increase the communication bandwidth. The paper focuses on general block-cyclic data distributions, i.e. data distributions used by parallelizing FORTRAN compilers [21]. Since block-cyclic distributions are very common, the communication between the distributed memory system and the network interface (including data reshuffling) was implemented as a library that can be used by applications.

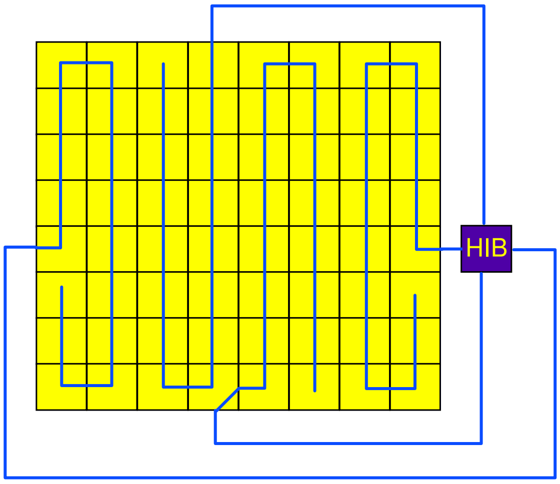

Figure 12: I/O configuration for block-cyclic data distributions

Figure 12 shows the connections used by the library for communicating with the network interface. Reshuffling converts the block-cyclic distribution used by the application into an "I/O" block-cyclic distribution in which each column of the system holds 8 Kbyte of contiguous data. Each pair of columns then sends a 16 Kbyte SU to the network interface where the data is merged to form a 64 Kbyte TU (i.e. striping over four links).

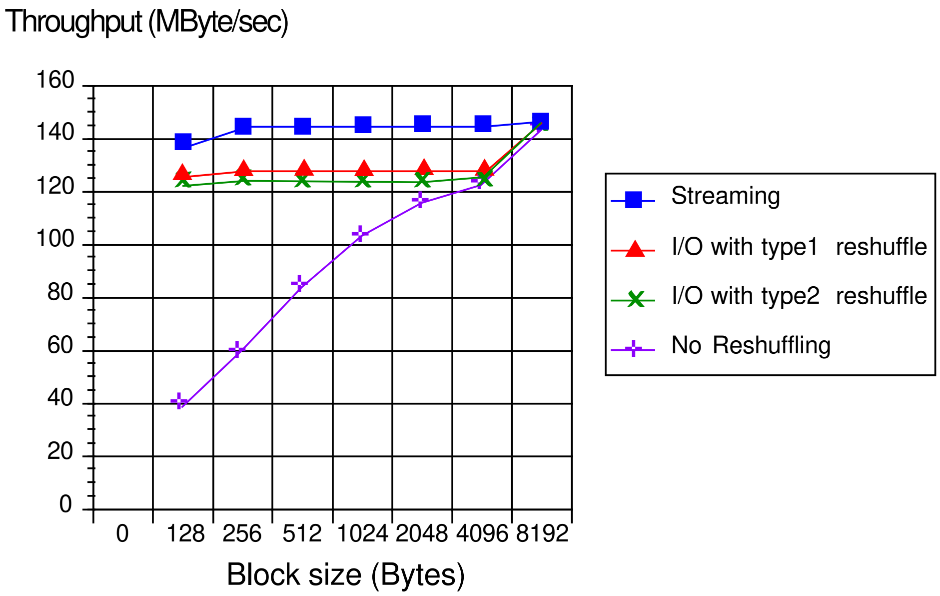

Figure 13: Throughput between network interface and system for block-cyclic data distributions

Figure 13 shows that we can achieve throughputs between the network interface and the system at rates close to the hardware bandwidth [5], even for small block sizes. An important feature is that the data movement was optimized for the specific data distribution.

5.4 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 [23]. 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 14 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 cameras, while 26 other cells reconstruct the 4 images and reshuffle the data. 4 interface cells are used to forward the four images to the framebuffer through the network interface.

The HIB with the *streams software* was able display four B/W images at a resolution of 486x496 pixels at 30 frames per second, which corresponds to a sustained throughput of close to 30 MBytes/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 MBytes/seconds. Due to limited memory resources on the iWarp, this cannot be achieved on the production system.

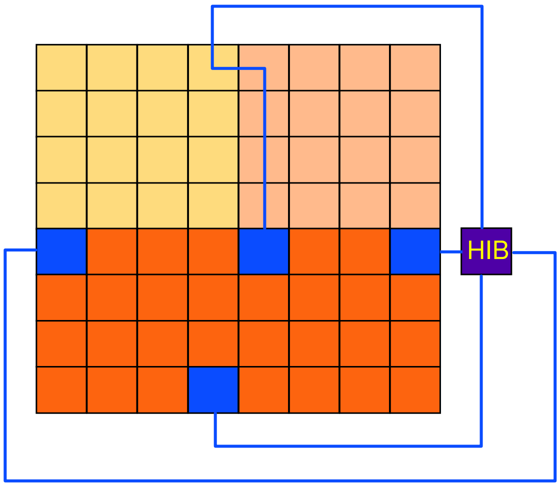

Figure 14: I/O configuration for digital VCR

6 Applicability to Other Systems

We looked at how the streams architecture can be used to support network I/O through the HIPPI interface of the Paragon system [13]. The Paragon is a distributed-memory system, like iWarp, but there are some significant differences in the architecture. Specifically, the Paragon uses a mesh interconnect and X-Y routing (iWarp has a torus and routing is under software control), has a link bandwidth that is higher than the HIPPI link speed (200 vs. 100 Mbytes/second), and has a full Unix operating system on each node (versus a light-weight runtime system on iWarp). As a result, striping can not be performed but the higher per-message overhead and routing restrictions increase the difficulty of handling fine-grain data distributions.

Even though the iWarp and Paragon architectures are quite different, the streams architecture readily applies. Specifically, the division of tasks between the distributed-memory system and the network interface, the separation of control and data interface, and the involvement of application in data transfer between the network interface and the system all carry over unchanged. This is in part a result of the fact that the streams architecture is driven by the overall architecture of distributed-memory systems (e.g. balance of power between network interface and the system) and application features (e.g. the use of different data distributions).

The main changes in an implementation of the streams architecture on the Paragon will be in the implementation of the data interface, the execution module and the control unit of the stream manager. The data interface will have to be reimplemented using the Paragon message passing model. This will also affect the execution module since it optimizes data transfers. For the stream manager, we can use threads to simplify the scheduling of the different modules on the network interface.

7 Conclusion

We achieved fast I/O rates with the iWarp in real-world applications using the *streams software*. 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 two key features in the streams architecture.

1. Application specific tasks are off-loaded from the network interface to the the distributed memory system. This minimizes the load on the network interface, and optimizes resource utilization across the entire application including I/O.

2. The *streams software* optimizes the network node's resource management and concentrates on efficient data transfers with a view of the whole distributed-memory system.

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 the *streams software* can provide the mechanisms to accomplish that task.

Acknowledgements

The research was supported by Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command (SPAWAR) under Contract N00039-87-C-0251.

We thank all the individuals from the iWarp and Nectar groups at CMU who have contributed to this project. Of special mention are Thomas Gross, Todd Mummert, Tom Warfel, Robert Clay, Chang-Hsin Chang, Doug Noll, Susan Hinrichs and Claudson Bornstein.

In addition we would like to acknowledge the use of computational resources provided by the Pittsburgh Supercomputing Center (PSC).

References

[1] 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.

[2] 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.

[3] 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.

[4] 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. Also published as CMU Technical Report CMU-CS-90-197.

[5] 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 Fransisco, August 1994. IEEE.

[6] 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.

[7] 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.

[8] 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.

[9] G.B. Dantzig. Planning under uncertainty using parallel computing. Technical Report SOL 87-1, Department of Operations Research, Stanford University, 1987.

[10] The MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883, Oregon, November 1993. ACM/IEEE.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Peter Steenkiste, Michael Hemy, Todd Mummert, and Brian Zill. Architecture and evaluation of a high-speed networking subsystem for distributed-memory systems. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*. IEEE, May 1994.

[20] 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.

[21] Jaspal Subhlok, Jim Stichnoth, Dave O'Hallaron, and Thomas Gross. Programming task and data parallelism on a multicomputer. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 13–22, May 1993.

[22] Richard Thompson. Los alamos multiple crossbar network crossbar interfaces. In *Workshop on the Architecture and Implementation of High Performance Communication Subsystems*. IEEE, February 1992.

[23] Jon Webb. Latency and bandwidth considerations in parallel robotics image processing. In *Proceedings of Supercomputing '93*, pages 230–239, Oregon, November 1993. ACM/IEEE.