A Parallel TCP/IP Protocol Implementation

Paul Mackerras David Sitsky

Department of Computer Science, Australian National University

Canberra ACT 0200 Australia

Abstract. With the ever-increasing network data rates available, the throughput achieved on a connection over a fast network using a transport protocol such as TCP/IP can easily be limited by the speed of the processor used to implement the transport protocol, rather than the network data rate. Distributing TCP/IP protocol processing over several processors in a distributed-memory parallel computer, the Fujitsu AP1000, is shown to provide a useful improvement in throughput achieved on connections over an FDDI ring.

1. Introduction

Modern network technologies are providing ever-increasing data rates. The FDDI (Fibre Distributed Data Interface) standard [1,2,3] specifies a data rate of 100Mbit/s, and ATM (Asynchronous Transfer Mode) networks [4] can operate at various data rates including 155Mbit/s, 622Mbit/s, and higher. Such high-speed networks are necessary if the large data sets used and produced by modern parallel supercomputers are to be transferred between machines in a reasonable time.

However, reliable transmission of data requires the use of some higher-level transport protocol, such as the Transmission Control Protocol (TCP) [5]. Transport protocols such as TCP are layered between the network interface driver software and application programs. Using TCP, applications can establish multiple independent connections with other applications. TCP is responsible for multiplexing and demultiplexing the data streams of the connections over the physical network connection. Other transport protocols exist, but are not widely supported.

Using a protocol such as TCP incurs overheads in processing time and in memory requirements. These overheads in processing time can easily exceed the processing capacity of a single processor at high data rates. Thus, if all of the processing required by the protocol implementation is performed on a single processor, the data transfer speed achieved will be limited to a fraction of the network data rate. Using several connections will not give any improvement, because the single protocol processor is the bottleneck. For example, a SPARC–1 processor running at 25MHz can perform TCP/IP protocol processing at 3–4MB/s. The fastest

processors available at present are perhaps 20 times faster; however, this is still not fast enough to utilize the full capacity of a gigabit-per-second link.

We assume we have a physical network with a data rate which is considerably in excess of the speed at which a single processor can perform all of the processing and data copying required by the transport protocol. We consider the situation of a distributed-memory parallel computer connected to such a network by physical network attachment hardware which is directly connected to one of the parallel computer's processors.

In the simplest configuration, we would have the transport protocol code running as a task on the processor which has the physical network attachment. Application code running on any processor could establish connections and send and receive data by sending appropriate messages (over the parallel machine's internal network) to the transport protocol task. Such a configuration would provide useful functionality, but with a lower performance (achievable data transfer speed) than one would expect either from the network data rate or from the aggregate processing speed of the parallel machine.

We could achieve a higher data transfer rate by using multiple network attachments, either to the same physical network (assuming a common-bus style of network), or to different physical networks. However, this is expensive in hardware terms, and would present problems in balancing the load and in coordinating the processing of packets for one connection over several processors. In addition, the use of multiple interfaces for a single connection is not supported by TCP/IP, and would thus require modified software to be installed on any other machine with which we wished to communicate.

Instead we have chosen to distribute the processing of data to/from a single network attachment over several of the processors in a parallel machine, since this potentially allows us to exploit the full bandwidth that the network is capable of. Of course, this approach can be combined with the use of multiple network attachments to achieve even higher bandwidth.

2. Characteristics of TCP/IP.

The processing and memory overheads for TCP/IP can be divided into three categories:

1. Processing the protocol header(s) on each incoming and outgoing packet uses CPU time.
2. TCP uses a checksum computed over the whole packet to verify that the protocol header and the data in each received packet have not been corrupted. Generating and checking this checksum consumes CPU time.
3. The data being sent or received on a connection will generally be copied twice; once between the application and a buffer maintained by the protocol code, and once between that buffer and the interface hardware. The buffers maintained by the protocol code consume memory, and the copying of the data consumes CPU time, although in some cases, DMA hardware can reduce this. (There are some implementations which can copy data directly between the application's buffer and the interface hardware (e.g. [6]), but these implementations require special hardware support.)

One of the most important factors affecting the processing overhead per byte of data transferred is the size of the data portion of each packet. Increasing the number of data bytes per packet reduces the protocol processing overhead per byte, because the per-packet overhead for processing the protocol header is amortized over a larger number of bytes.

With TCP, the number of data bytes per packet is controlled by the maximum segment size (MSS) parameter, which is the maximum number of data bytes which may be transmitted in one packet. The MSS is limited by the maximum transmission unit (MTU) of the network interface being used to transmit the packets. The MTU is the size of the largest packet that the interface can transmit or receive, excluding any link-level headers. For an FDDI interface, the MTU is usually

4352. The MTU is limited in turn by the maximum FDDI frame size of 4500 bytes including all link-level and MAC (medium access control) headers. We have used a MSS of 4096 bytes in all experiments.

Another factor which significantly affects the throughput achieved over a TCP connection is the *window size*. TCP uses a sliding-window scheme for flow control, in which the receiver controls the flow of data from the sender by specifying how many bytes of data it is prepared to accept from the sender. This amount of data is called the window. Ideally, the window size should be no less than the product of the network bandwidth and the round-trip latency—the time taken for a packet of data to travel to the receiver and the corresponding acknowledgement to return to the sender. A window size which is smaller than this will result in reduced throughput, because the sender will waste time waiting for acknowledgements or window updates from the receiver.

3. Experimental hardware environment

The experiments described below were carried out on the Fujitsu AP1000 at the Australian National University (ANU). The Fujitsu AP1000 is a parallel computer with 64–1024 processors or *cells*, each of which has a SPARC-1 CPU running at 25MHz with 16MB of memory. The AP1000 at ANU has 128 cells. The AP1000 is controlled by a host computer. The connection to the host has a relatively low bandwidth; in practice, data transfers to and from the host occur at no more than 1MB/s. Other than the FDDI interface described below, the host interface is the only means of communicating with other computers. Since the total memory of the AP1000 at ANU is 2GB, and data sets in use on this machine reach 4GB, the slow speed of the host interface is a significant restriction.

One of us (Mackerras) has recently designed and constructed an FDDI interface for the AP1000 at ANU. The purpose of constructing the FDDI interface was to allow the AP1000 to communicate with other high-performance computers and mass-storage servers on the ANU campus. FDDI was chosen because most of these machines were already connected to an existing FDDI ring. These machines all run the Unix operating system and support TCP/IP as the principal (or only) protocol for reliable stream-oriented network connections.

4. Design

We performed some preliminary experiments to determine how fast we could expect an AP1000 cell to perform the processing necessary to transfer data using TCP/IP. We set up two cells of the AP1000 as two separate hosts, connected via a virtual network implemented using the native communications library primitives of the AP1000. The first cell established a connection to the second and sent 10MB of data. The results showed that, of a total of 4.0 seconds taken to transfer the 10MB of data, 3.2 seconds were attributable to the data-handling aspects of the processing (copying and checksumming the data). Thus the protocol header processing took no more than 0.8 seconds.

The data-handling parts of the processing (that is, copying and checksumming the data) are quite simple and can easily be parallelized over several cells. The processing of the protocol headers is not so easily parallelized. However, the results from the preliminary experiments indicate that the protocol header processing overhead is relatively small compared to the data handling overhead. Thus we can obtain a speedup of several times by running the protocol processing on one cell and distributing the data handling over several other cells. It is relatively

easy to separate the protocol processing from the data handling because the protocol processing is almost completely independent of the specific data being transmitted.

Thus the processing workload is divided among three types of cells (see figure 1):

1. The *interface* cell has the physical network hardware attached. It is responsible for receiving packets from the network and sending them to the data cells, and for gathering packets from the data cells and transmitting them on the network. There is one interface cell for each network interface in the system.
2. The *data* cells handle all the data-intensive parts of the processing. They calculate the checksums for received and transmitted packets, transfer data to and from the application programs running on other cells, collect data together to construct packets for transmission, separate the header and data sections of received packets, and send the received headers to the protocol cell. There are many data cells, typically 5 or more.
3. The *protocol* cell handles all the details of the TCP/IP protocol, such as determining when to send data or acknowledgements to the peer, and when to release data to the application programs. It also handles all socket-related systems calls from the application programs, and manages a pool of memory on the data cells. There is at present only one protocol cell, although it would be possible (and desirable) to distribute its workload over several cells (see below).

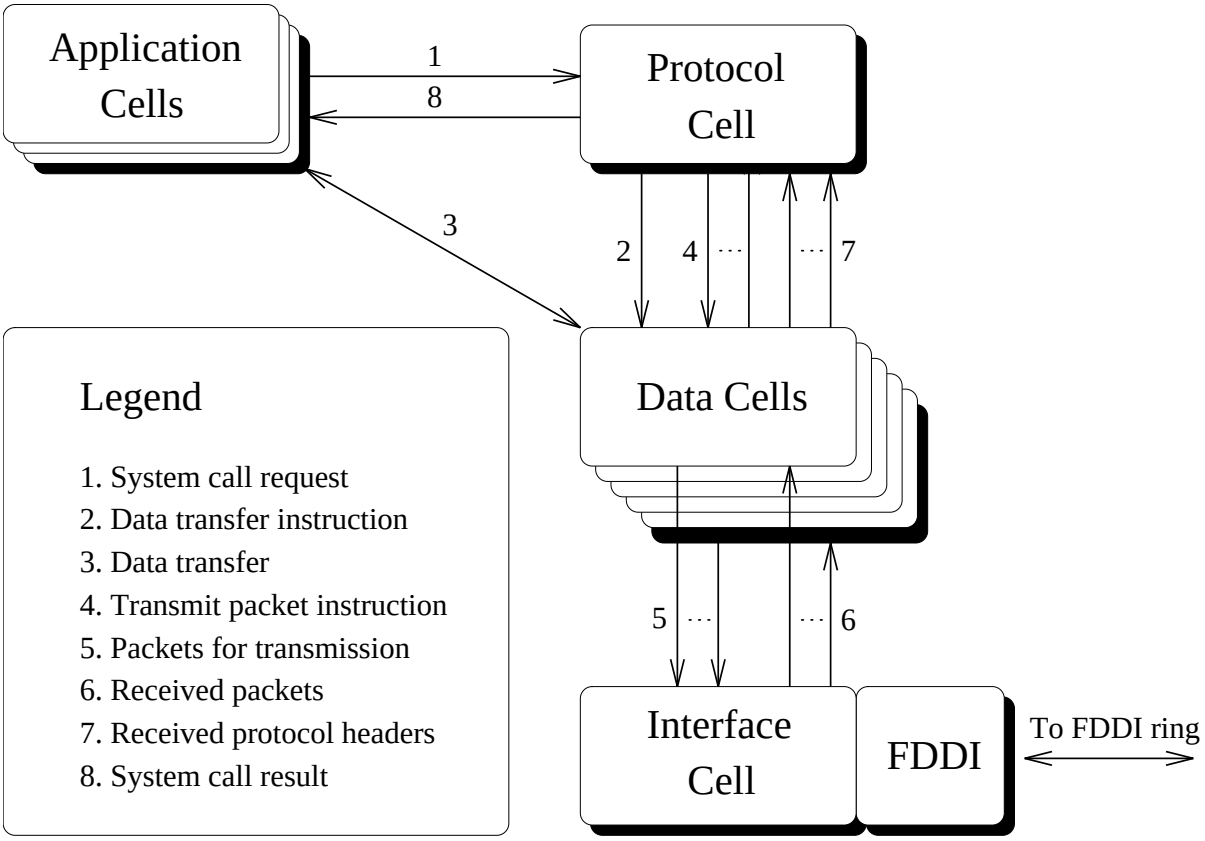

Figure 1. Architecture of distributed implementation

Typically, we would have between 4 and 10 data cells. The TCP/IP protocol processing thus uses between 6 and 12 cells of the 128 in the AP1000 at ANU. Application tasks run on the remaining cells. Due to limitations in the cell operating system, the protocol, interface and data cells all run a continuous polling loop, leaving no processor time available for other tasks on those cells.

We have arranged our implementation so as to make use of the `put` and `get` system calls on the AP1000. These calls transfer data directly from one cell's memory to another using the direct-memory access (DMA) hardware and without involving any extra copying of the data. These calls are the fastest way to transfer large quantities of data from one cell to another; however, they require that the cell which initiates the operation knows the address of the data in the other cell.

Processing of the data in one packet is performed by one data cell, not split over several data cells. Parallelism is obtained by having different data cells process successive packets. This approach works well because TCP allows the sender to send several packets ahead of the acknowledgements received (depending on the window size, which we set as large as possible). Given the overheads involved in sending data between cells, it is important to keep the amount of data communicated with each call as large as possible; that is, it is less efficient to send (say) 8 blocks of 512 bytes to different cells than it is to send a block of 4096 bytes to one cell.

The protocol cell is responsible for managing a pool of memory on the data cells. This pool is managed using an extension of the *mbuf* data structure used in the BSD network code. We have defined a new type of mbuf which we call a ''remote mbuf''. A remote mbuf uses the mbuf data structure and is manipulated on the protocol cell like ordinary mbufs, except that the associated data is stored on another cell. The mbuf data structure on the protocol cell contains only the cell number, address and length identifying where the data is stored on the remote cell.

5. Results

We measured the throughput achieved over one or more connections with this design by measuring the time taken to transmit a quantity of data (typically 10MB) from one application to another, where one application was running on an AP1000 cell and the other on a workstation which was also attached to the same FDDI ring. The applications we used were very simple test programs which did not read or write any data to disk; they simply established the connection, allocated a large buffer (typically 64k bytes) and read (or wrote) this buffer from (to) the socket file descriptor several times.

For the tests which measured the aggregate throughput achieved over several (say N) connections, we used N application tasks running on different AP1000 cells, each communicating with an application running on N different workstations attached to the FDDI ring. The workstations were all SparcStation 10 or 20 machines. The application tasks running on the AP1000 synchronized before and after the transfer. The aggregate throughput was measured as the total amount of data transferred divided by the time taken from the start of the transfers until after the tasks had synchronized after the transfers.

Figures 2 and 3 show the throughput achieved for 1, 2 and 3 simultaneous connections, as a function of the number of data cells. Figure 2 shows the results for the AP1000 receiving data from the workstation(s), while figure 3 shows the results for the AP1000 sending data to the workstation(s).

It is evident that, provided we use at least 5 data cells, the throughput is not limited by the speed of the data cells. For two or more connections, it is evident that the throughput is not limited by the speed of the workstations, since the workstations are clearly capable of at least 5 MB/s each. Clearly either the protocol cell or the interface cell is limiting the throughput.

To test whether the speed of the protocol cell was limiting the throughput, we measured the proportion of time that the protocol cell was idle during the transfer, for each of the data points in figures 2 and 3. These results are shown in figures 4 (for the AP1000 receiving data) and 5 (for the AP1000 sending data). The results indicate that the protocol cell is the bottleneck when the AP1000 is receiving data, but not when it is sending data.

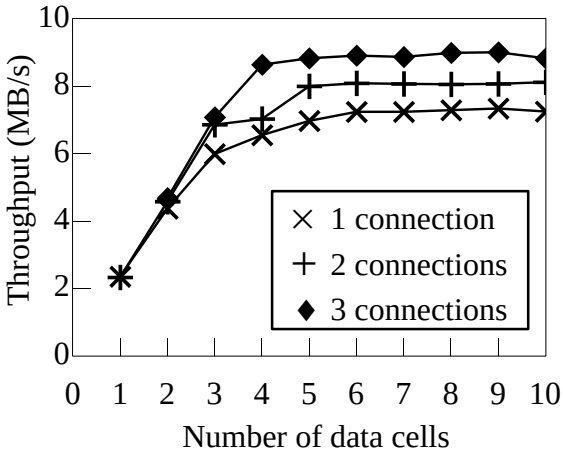

Figure 2. Aggregate throughput achieved with the AP1000 receiving data, as a function of the number of data cells.

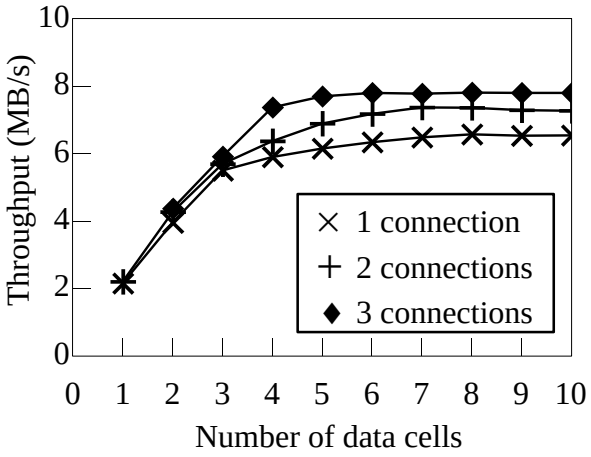

Figure 3. Aggregate throughput achieved with the AP1000 sending data, as a function of the number of data cells.

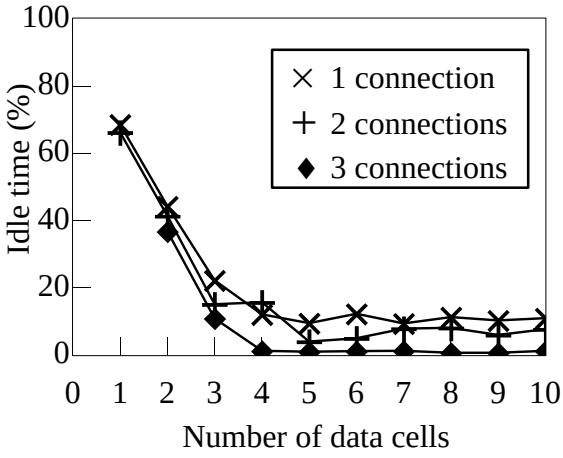

Figure 4. Idle time for the protocol cell with the AP1000 receiving data, as a function of the number of data cells.

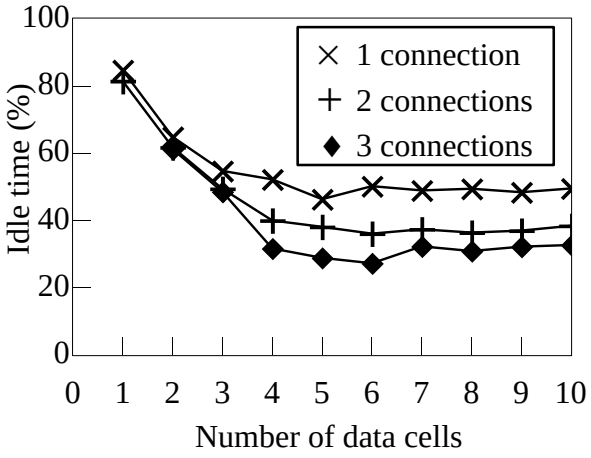

Figure 5. Idle time for the protocol cell with the AP1000 sending data, as a function of the number of data cells.

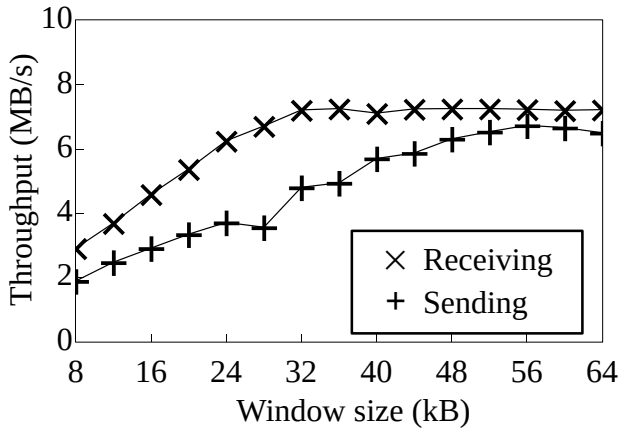

Figure 6. Throughput for a single connection as a function of window size.

Further investigations showed that the speed at which the interface cell can `get` packets from the data cells is the bottleneck when the AP1000 is sending data. The interface cell had one or more outstanding `get`s for more than 90% of the time while it was sending data at around 7.5MB/s.

Figure 6 shows the results of measuring the throughput achieved on a single connection as a function of the window size. The results show that for window sizes of 48k bytes or more (as used in all of the other experiments), the window size is not the factor limiting the throughput.

6. Discussion

This design was reached after several previous designs were implemented and analysed for bottlenecks. In particular, we found it necessary to keep the processing on the interface cell to an absolute minimum, because it handles all of the data. Its basic functions cannot be parallelized, because there is only one physical network attachment. An earlier design had the interface cell splitting received packets into header and data sections, sending the header section to the protocol cell, and sending the data section to one of the data cells. Also, the interface cell was responsible for collecting together pieces of packets to be transmitted from the data cells and putting them together with a header received from the protocol cell to make a complete packet. With this design, we found that the overall throughput was severely limited by the interface cell.

Even with the interface cell only handling complete packets, it is still the bottleneck when the AP1000 is sending data. More particularly, the latency of the `get` operation is the limiting factor. Although the bandwidth of the `get` operation is quite high for large transfers (of the order of 15MB/s), the `get`s performed by the interface cell only achieve about half of this figure because of their relatively small size (approximately 4kB, limited by the maximum packet size.)

As the protocol cell speed is the limiting factor when the AP1000 is receiving data, it is desirable to parallelize its functions. One way that the operations of the protocol cell could be parallelized would be to process different connections on different protocol cells. This would allow a higher total throughput over several connections. To do this, it would be necessary to provide for the data structures describing a connection to be passed from one protocol cell to another, both when a connection is first established, and possibly at a later time for load balancing. It would also be necessary to maintain data structures on the data cells so that they could forward the headers of received packets to the correct protocol cell, according to which connection they were for.

The transfer rate on a single connection increases with increasing window size up to a window size of approximately 48kB. Since the maximum window size is 64kB, it is clear that with improvements to the other parts of the system, the window size may well become the limiting factor. The BSD-4.4 network code we have used includes support for the TCP large window option [7]; unfortunately, none of our workstations have kernel support for this option.

In order to compare the speed of the parallel implementation with a serial implementation on the same processor hardware, we implemented a system which performed all of the functions of the interface, protocol and data cells on one cell (the cell with the FDDI interface). As before, the application task(s) ran on other cells. With this single-cell implementation, we measured a throughput of 3.1 MB/s over a single connection with the AP1000 receiving data, and 3.5 MB/s with the AP1000 sending data. Thus, the parallel implementation is faster than the serial implementation for four or more processors (two or more data cells), and achieves a speedup over the serial implementation of 2–3 if seven or more processors are used. This is a worthwhile speed increase in terms of making use of the available bandwidth of the FDDI ring.

7. Conclusion

Our implementation demonstrates that it is possible to distribute the processing of a transport protocol over several processors and obtain a worthwhile speedup. The speedup that we have achieved enables us to make much better use of the available bandwidth of the FDDI ring. Although the AP1000 has relatively slow processors by modern standards, neither is the FDDI speed of 100Mbit/s at the leading edge of current network technologies. We believe that the techniques we have developed will continue to be useful with fast modern processors as network speeds escalate to 1Gbit/s and beyond.

8. References

[1] ISO Standard 9314-1: 1989 (E), ‘‘Information processing systems—Fibre Distributed Data Interface (FDDI)—Part 1: Token Ring Physical Layer Protocol (PHY)’’, 1989.

[2] ISO Standard 9314-2: 1989 (E), ‘‘Information processing systems—Fibre Distributed Data Interface (FDDI)—Part 2: Token Ring Media Access Control (MAC)’’, 1989.

[3] ISO Standard 9314-3: 1989 (E), ‘‘Information processing systems—Fibre Distributed Data Interface (FDDI)—Part 3: Physical Layer Medium Dependent (PMD)’’, 1989.

[4] A. Hac and H.B. Mutlu, ‘‘Synchronous Optical Network and Broadband ISDN Protocols,’’ *Computer* **22**(11), 26--34, November 1989.

[5] Postel, J., ‘‘Transmission Control Protocol,’’ Request for Comments 793, Network Working Group, Internet Network Information Center, 1981.

[6] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K. Su, ‘‘Myrinet: A Gigabit-per-Second Local Area Network,’’ *IEEE Micro* **15**(1), 29–36, February 1995.

[7] V. Jacobson, R. Braden, and D. Borman, ‘‘TCP Extensions for High Performance,’’ Request for Comments 1323, Network Working Group, Internet Network Information Center, May 1992.