Performance Results for a Reliable Low-latency Cluster Communication Protocol

Stephen R. Donaldson[1,2], Jonathan M.D. Hill[2], and David B. Skillicorn[3]

[1] Oxford University Computing Laboratory, UK.
[2] Sychron Ltd, 1 Cambridge Terrace, Oxford, UK.
[3] CISC, Queen's University, Kingston, Canada

Abstract. Existing low-latency protocols make unrealistically strong assumptions about reliability. This allows them to achieve impressive performance, but also prevents this performance being exploited by applications, which must then deal with reliability issues in the application code. We present results from a new protocol that provides error recovery, and whose performance is close to that of existing low-latency protocols. We achieve a CPU overhead of $1.5\mu s$ for packet download and $3.6\mu s$ for upload. Our results show that (a) executing a protocol in the kernel is not incompatible with high performance, and (b) complete control over the protocol stack enables (1) simple forms of flow control to be adopted, (2) proper bracketing of the unreliable portions of the interconnect thus minimising buffers held up for possible recovery, and (3) the sharing of buffer pools. The result is a protocol which performs well in the context of parallel computation and the loose coupling of processes in the workstations of a cluster.

1 Introduction

In many respects, recent results from the low-latency communication community are analogous to drag racing. There is a pre-occupation with acceleration and top speed (latency and bandwidth) at the expense of general useability, in the same way that dragsters race on straight stretches of road (point-to-point benchmarks) in which crude braking and steering mechanisms suffice (no error recovery and poor security). Some of the performance claims that clusters of PCs outperform commercial parallel machines are therefore misleading, because the impressive performance results do not necessarily transfer to more general settings.

In this paper we present "dragster style" micro-benchmark results of a low-latency communication protocol used in an implementation of *BSPlib* [9, 15]. However, unlike much other work, the protocol deals fully with reliability issues, and can be used "as is" by application programs. Like other work in this area, we replace existing heavyweight protocol stacks with a lightweight protocol, using a purpose-built device driver to support low-latency communication in a cluster of PCs connected by 100Mbps switched Ethernet. Unlike most other low-latency protocols, we provide the necessary functionality of protocols such as TCP/IP to support SPMD-style parallel computation among a group of processors. The

effectiveness of the complete system is tested using the NAS parallel benchmarks. Our results show that for a modest budget of $US12,000, an eight-processor BSP cluster can easily outperform an IBM SP2 (1995), and has performance similar to an SGI Origin 2000 (O2K) (1998) *as observed by application programs.*

One of the issues addressed by this paper is whether error recovery should be handled in user or kernel space. Reliability is not orthogonal to performance, and cannot be designed in later. Protocols that ignore error recovery, such as U-Net [29], GAMMA [7], and BIP [20], are based upon the premise that the link error probability is of the same order as processor failure. This assumption is true when the link is used for point-to-point micro-benchmarks. However, for general patterns of communication between a collection of processors connected by a switch (which is the only sensible *scalable* interconnect) there is the potential for congestion, which leads to dropped packets within the switch. This is indistinguishable from hard link errors. For example, if several processors send a large amount of data to a single processor then, unless some back-pressure notification is made available to the sender, it is always possible to overwhelm a component of the switch.

In our implementation of *BSPlib*, we use a three-tiered approach to providing a general-purpose communication system. At the highest level, we provide the user with the *BSPlib* API, which is implemented on top of a reliable communication middle layer that provides primitives for handshaking and initialisation of the computation on the various processors, mechanisms for orderly and unorderly shutdown, a nonblocking send primitive, a blocking receive primitive and a probe to test for message presence. This middle layer is itself implemented on top of a transport protocol that provides the necessary functionality to support acknowledgement of packets, error recovery and flow control. The lowest layer implements low-latency point-to-point datagram communication by using a kernel module/device driver that services queues that are also manipulated by the user process in user space. By using a network interface card that automatically polls for work when a queue of send descriptors dries up, it is possible to schedule packet transmission without the need for a system call.

We claim that, while the community has concentrated on the "drag-racing" layer, this is neither particularly large (less than 10% of the total code of *BSPlib*), nor difficult to implement. In contrast, the error-recovery component, which has been ignored by many researchers, is considerably more complex and significantly larger. The problems that need to be addressed in the error-recovery layer are: (1) queueing of outgoing packets in case they are dropped; (2) acknowledging packets on send queues; (3) detecting dropped packets at receivers; and (4) initiating retransmission. All of this extra functionality bloats the protocol stack, which affects the performance of micro-benchmarks because packet download consumes more CPU cycles and asynchronous packet upload may incur long running interrupt handlers which upload messages "off the wire." We investigate three different implementations of *BSPlib*: (1) a standard TCP/IP variant in which error recovery is sender-based and performed by TCP within the kernel; (2) one in which error recovery is receiver-based and is performed in user space

by *BSPlib*; and (3) one in which error recovery is receiver-based and is performed within the kernel by a BSP device driver. Comparing the three approaches, we demonstrate that a receiver-based protocol is superior to a sender-based one, and that handling recovery in the kernel is better than handling it in user space.

Our solution compares favourably to SP2 and O2K machines with proprietary protocols and to MPI on the cluster. We illustrate this with both micro-kernel benchmarks and the NAS parallel benchmarks.

2 Network messaging layers for *BSPlib*

BSP [23, 25] programs written using the *BSPlib* API [15] consist of a series of *supersteps* which are global operations of the entire machine. Each superstep consists of three sequential phases of: (1) computation, (2) communication,and (3) a barrier synchronisation marked by all processors calling `bsp_sync()`.

BSPlib is implemented on top of a variety of lower-level messaging layers. In this paper we describe three implementations of the messaging layer: (1) the BSPlib/TCP implementation (Section 2.1) that uses BSD stream sockets as an interface to TCP/IP; (2) the BSPlib/UDP implementation (Section 2.2) that uses BSD datagram sockets as an interface to UDP/IP; and (3) the BSPlib/NIC implementation (Section 2.3) that uses the same error recovery and acknowledgement protocol used in the BSPlib/UDP implementation, but replaces the BSD datagram interface with a lightweight packet transmission mechanism that interfaces directly with the network interface card. In all the implementations, any *BSPlib* communications posted during a superstep are delayed until the barrier synchronisation that marks the end of the superstep (as this is a clear performance win [16]). The actual communication in *BSPlib* therefore reduces to the problem of routing a collection of packets between all the processors, within the `bsp_sync()` procedure that marks the end of the superstep.

2.1 BSPlib/TCP: a messaging layer built upon TCP/IP

The BSD stream socket interface provides a reliable, full-duplex, connection-oriented byte stream between two endpoints using the TCP/IP protocol. As well as providing error recovery to deliver this reliability, TCP/IP uses a sliding window protocol for flow control.

General and reliable transport protocols such as TCP/IP provide a portable implementation path for BSP. However, the functionality of TCP/IP is too rich for BSP-style computation using a dedicated set of processors. For example, the TCP/IP protocol stack cannot take advantage of the nature of the local LAN as the stack also includes support for long-haul traffic, where nothing may be known about the intermediate networks. This makes TCP/IP unsuitable for high-performance computation [10].

BSPlib/TCP uses the `send()` function to push data into the TCP/IP protocol stack. At the level of the messaging layer, reception of messages is *not* asynchronous. When the higher level requires a packet, the messaging layer waits for a packet to arrive at the process by issuing a `select()`. When the `select()`

completes, the received packet is copied into user space by issuing a `recv()`. Of course, message reception is still asynchronous with respect to the lower levels of the TCP/IP protocol stack. However, messages will be buffered within the stack until they have been selected by the messaging layer. Having the actual reception at a lower level makes it particularly difficult for the messaging layer to make any sensible decisions about buffering. For example, if packets are dropped due to insufficient buffer resources, the upper layer is unaware of it, and cannot plan to circumvent the problem by, for example, using global knowledge of the communication pattern. The only way of avoiding an excessive number of packets being dropped by the TCP/IP layer is to associate large buffers with each of the $p-1$ sockets that form the endpoints on each process ($p(p-1)$ in total). This may be an extremely wasteful use of memory if an application uses only skewed communication patterns.

Error recovery in point-to-point protocols over unreliable media is typically accomplished by using timeouts and acknowledgement information that flows back from the receiver to the sender. To improve the usage of the media, TCP/IP attempts to piggy-back such acknowledgements on traffic flowing in the reverse direction on the circuit. In order to do this, acknowledgements may be delayed for a short period in the hope that reverse traffic will be presented for transmission and upon which the acknowledgement can be piggy-backed. Should no reverse traffic be forthcoming within $200ms$, an explicit acknowledgement packet is returned. From the senders point of view, should no acknowledgement be forthcoming for a specified time (usually $1.5s$, with an exponential back-off up to $64.0s$), a timeout will trigger at the sending station which then assumes that either the data was dropped en route or that the acknowledgement (explicit or piggy-backed) was dropped on the return path. In either case, TCP/IP retransmits the data from the point after the last received acknowledgement (a form of go-back-n protocol). These timeouts are quite large as TCP/IP is designed for the long haul traffic hence is not really suitable for intensive bursts of communication between a tightly-coupled group of machines on a fast network.

2.2 BSPlib/UDP: a messaging layer built upon UDP/IP

The BSD datagram socket interface provides an unreliable, full-duplex, connectionless packet delivery mechanism between machines using the UDP/IP protocol. Unlike TCP streams, UDP datagrams have a fixed maximum transmission unit size of 1500 bytes for Ethernet frames. The link between the machines is unreliable as UDP/IP provides no form of message acknowledgement, error recovery, or flow control.

Protocols such as UDP/IP have a shallower protocol stack and therefore have more potential for high performance. However, they do not provide reliable delivery of messages. If user APIs such as *BSPlib* are implemented over UDP/IP, they must handle the explicit acknowledgement of data and error recovery themselves. In the protected kernel/user-space model of UNIX, this must be done in user space.

To implement a reliable send primitive, BSPlib/UDP copies a user data structure referenced in a send call into a buffer that is taken from a free queue. If there are no free buffers available, then the send fails, and it is up to the higher level *BSPlib* layer to recover from this situation[4]. The buffer is placed on a send queue and a communication is attempted using the `sendto()` datagram primitive. Queueing the buffer on the send queue ensures that it does not matter if the packet communicated by `sendto()` fails to reach its destination, as our error-recovery protocol can resend it if necessary. Only when a send has been positively acknowledged by the partner process are buffers reclaimed from the send queue to the free queue.

Message delivery is asynchronous and accomplished by using a signal handler that is dispatched whenever messages arrive at a processor (SIGIO signals). Within the signal handler, messages are copied into user space by using `recvfrom()`. To perform the error recovery as soon as possible, it executes from within the signal handler in *user space*.

Recall that the TCP/IP error recovery protocol is too simplistic for high-performance use in *BSPlib* due to its go-back-n error recovery policy, which may generate a large number of duplicate packets, and its slow-acting timeout mechanisms. These problems are solved in BSPlib/UDP by implementing our own error recovery mechanism that uses a selective retransmission scheme that only resends the packets that are actually dropped, and a sophisticated acknowledgement policy that is based upon available buffer resources and not timeouts [9, 22]. The acknowledgement scheme has both a sender and receiver component to the algorithm. From the sender's perspective, if the number of buffers on a free queue of packets becomes low, then outgoing packets are marked as requiring acknowledgements. This mechanism is triggered by calculating how many packets can be consumed (both incoming and outgoing) in the time taken for a message roundtrip. The sender therefore anticipates when buffer resources will run out, and attempts to force an acknowledgement to be returned just before buffer starvation. This strategy minimises both the number of acknowledgements, and the time a processor stalls trying to send messages. From the receiver's viewpoint, if n is the total number of send and receive buffers, then a receiver will only send a non-piggybacked acknowledgement if it is was requested, and at least $\frac{n}{2p}$ elements have come in over a link since the last acknowledgement (either piggy-backed or explicit).

2.3 BSPlib/NIC: a NIC-based transport layer for *BSPlib*

A Network Interface Card (NIC) provides the hardware interface between a system bus and the physical network medium. BSPlib/NIC uses exactly the same error-recovery and acknowledgement protocol as BSPlib/UDP. The only difference between the implementations is that BSPlib/NIC is built upon a lightweight

[4] *BSPlib* recovers by either: (1) receiving any packets that have been queued for *BSPlib*; or (2) asking those links that have a large number of unacknowledged send buffers to return an acknowledgement.

packet transmission mechanism that interfaces directly to the NIC. This is achieved by having a portion of memory used for communication buffers and shared data structures that are mapped into both the kernel and the user's address space.

With the sophistication of modern network interface cards, UDP-like protocols can be implemented to achieve very high bandwidth utilisation of the network. All outgoing packets contain a protocol header which contains piggy-backed acknowledgement and error recovery data. As the *BSPlib* layer can queue a potentially large number of packets for transmission by the NIC, the protocol information may become stale as incoming packets update the protocol state. This problem can be alleviated by allowing the NIC to perform a gathered send, whereby the protocol information is only read when the packet is about to be transmitted. Processor usage is also improved by using the features of the NIC to make the communication as asynchronous as possible. For example, the standard technique in device drivers is to use bounded send and receive descriptor rings which the NIC traverses. In our implementation, the NIC traverses an arbitrary sized (linked-list) queue of buffers (the NIC's transmit queue).

In the NIC implementation, the user-space signal handler that was used for asynchronous message delivery in BSPlib/UDP is moved into the kernel as a interrupt handler associated with the interrupt request line used by the NIC. A sequence of receive buffers are serviced by the NIC, such that upon successful packet upload, an interrupt is triggered, and the handler *swaps* the newly-uploaded buffer with a fresh user space mapped buffer from the free queue (this eliminates a memory copy). The handler then inspects the uploaded packet to determine if any error recovery should be triggered *within the kernel*. Finally the packet is placed on a receive queue that can be manipulated in both kernel and user space. As this receive queue is manipulated by the user process, it eliminates the need for a system call, but slightly complicates queueing, as the queue data structure can be concurrently accessed by either the kernel or user process.

3 Benchmarks

We have benchmarked our work at various levels: at the lowest level, micro-benchmarks show the raw efficiency of the communication equipment that can be achieved. At a higher, application level we compare our cluster to other popular computers. For these benchmarks we use a *BSPlib* port [17] of the NAS Parallel Benchmarks 2.1 [2]. To compare the efficiency of the cluster and *BSPlib* with other parallel machines, we also include results from the MPI versions of the benchmarks. On the SP2 (66MHz thin-node) and the O2K, results are for proprietary implementations of MPI; on our cluster they are for `mpich` [14].

3.1 Cluster configuration

The prototype system is a cluster of eight 400MHz Pentium II PC systems each with 128MB of 10ns SDRAM on 100MHz motherboards. Two distinct types

of communication are required: slow(er) I/O communication and fast computation data. It simplified coding and debugging to have two separate networks; a control network to deal with I/O, and a network reserved for interprocess application data. The control network uses the standard TCP/IP protocol suite. We concentrate on the data network.

Each of the processors runs the Linux 2.0 Kernel and the driver software is written according to the interfaces documented in [21]. Other than reserving memory for communication, no kernel changes are required. The communication devices used were eight 3COM 3C905B-TX NICs running at 100Mbps [1] and a 100Mbps Cisco 2916XL fast Ethernet switch [8].

3.2 Micro-Benchmarks

The micro-benchmarks are typical and measure the raw bandwidth and latency that can be achieved 'on the wire' (excluding protocol data). We consider three classes: the roundtrip delay between two processors for various message size (Figure 1); link bandwidth between two processors for various message size (Figure 2); and per-packet latency for half-round-trip packets. In this last class, we provide results for short (4 byte) (Figure 3) and large (1400 byte) (Figure 4) packet sizes, and for various number of messages. These benchmarks use the following configurations:

Key	Machine	Library	Transport	Network
PII-BSPlib-NIC-100mbit-wire	PII Cluster	*BSPlib*	NIC	100BASE-TX, cross-over wire
PII-BSPlib-NIC-100mbit-2916XL	PII Cluster	*BSPlib*	NIC	100BASE-TX, Cisco 2916XL.
PII-BSPlib-UDP-100mbit-2916XL	PII Cluster	*BSPlib*	UDP/IP	100BASE-TX, Cisco 2916XL
PII-BSPlib-TCP-100mbit-2916XL	PII Cluster	*BSPlib*	TCP/IP	100BASE-TX, Cisco 2916XL
PII-MPI-ch_p4-100mbit-2916XL	PII Cluster	mpich	ch_p4	100BASE-TX, Cisco 2916XL
SP2-MPI-IBM-320mbit-vulcan	IBM SP2	IBM's MPI		320 Mbps Vulcan
SP2-MPL-IBM-320mbit-vulcan	IBM SP2	IBM's MPL		320 Mbps Vulcan
O2K-MPI-SGI-700mbit-ccnuma	SGI O2K	SGI's MPI		

Round-trip time: Figure 1 shows the roundtrip time between two processors for increasing message sizes. The experiment performed thousands of samples, and the *minimum* round trip delay time was recorded. This choice of minimum value accounts for the TCP/IP and UDP/IP results being identical since, when no packets are dropped and error recovery is not initiated, the two protocols behave similarly. The difference of approximately $100\mu s$ for small messages between the NIC and UDP/IP results demonstrates the shallowness of the protocol stack. Although this improvement is partly due to the elimination of the system calls, the major benefit is due to the replacement of the user-space signal handler that was used for asynchronous message delivery in BSPlib/UDP, with a kernel-space interrupt handler associated with the interrupt line serviced by the NIC.

As well as benchmarking the NIC-based protocol through a 100Mbps switch, two machines were connected back-to-back using a crossover wire. The purpose of this experiment was to identify the software latencies in the NIC, ignoring the effect of the switch. For 4-byte user messages (which will be 40 bytes including headers, but 60 bytes on the wire due to a 60-byte minimum frame size),

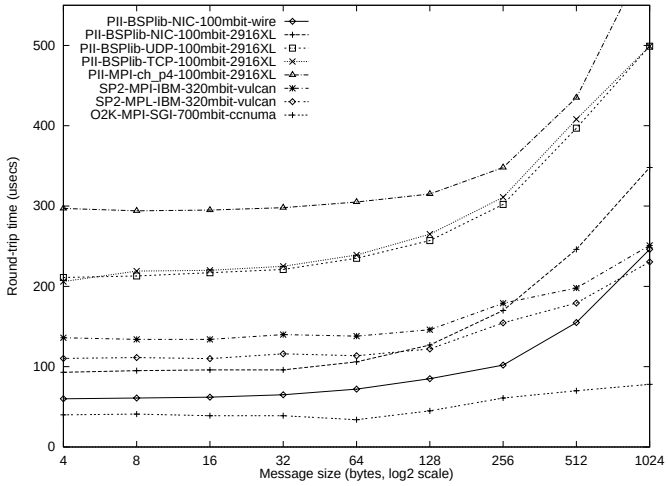

Fig. 1. Round-trip time between two processors as a function of message size

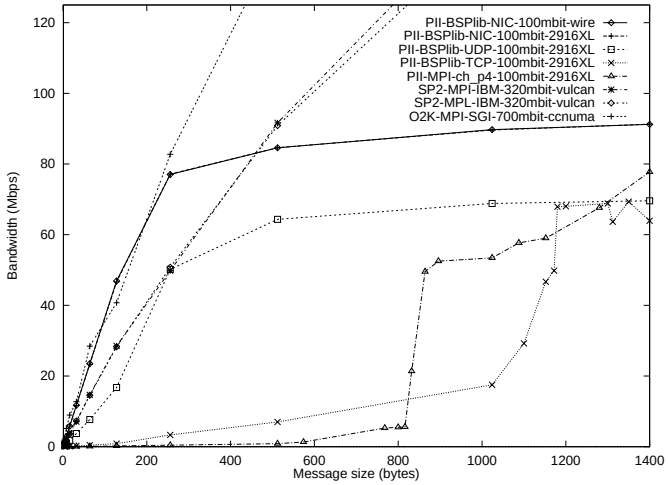

Fig. 2. Link bandwidth between two processors as a function of message size

the experiment PII-BSPlib-NIC-100mbit-wire achieved a roundtrip time of $58\mu s$. This compares to $93\mu s$ for the PII-BSPlib-NIC-100mbit-2916XL experiment, the difference being entirely accounted for by the Cisco 2916XL switch latency [18] of $2 \times 17.2 = 34.4\mu s \approx 93 - 58 = 35\mu s$.

From the figure, the latency of the NIC-based protocol running on the cluster is significantly smaller than any other protocol on the cluster, and outperforms the SP2 for messages less than 256 bytes. The loss in latency for larger messages is entirely due to the increasing latency through the switch. These results are consistent with, though slightly better than, the results of U-Net over 100Mbps Ethernet [29], *even with the overhead of queuing outgoing packets in case they are dropped and checking whether recovery should be triggered.*

Bandwidth: Figure 2 shows the sustained bandwidth between two processors for increasing message sizes. The benchmark arranges that one process sends

a large number of packets (with their sizes shown on the horizontal axis) to another process, which returns a single small packet after all the packets have been received. The bandwidth on the vertical axis is calculated from the amount of data communicated, and the time between the first packet sent and the small control packet returned. The return latency can be ignored as there are many packets sent by the first processor. However, the traffic includes any reverse communication required to acknowledge the received packets or to recover from errors.

Considering that the peak bandwidths of the O2K and the SP2 are much higher than that of a 100Mbps switch, it is surprising that the cluster can outperform the SP2 for user messages up to 400 bytes, and can match the O2K for messages up to 200 bytes. After these two points, these more-expensive machines perform considerably better.

The BSPlib/UDP and BSPlib/NIC transport layers, which both use the same error recovery and acknowledgement protocol [9], have smooth, predictable bandwidth curves that rise to their asymptotic levels quite early. In contrast, the BSPlib/TCP transport layer and `mpich` show erratic and late-rising curves. This demonstrates the assertion of the unsuitability of TCP/IP for high-performance computation due to its inappropriate acknowledgement and error-recovery mechanisms. Although this result would suggest that it is worthwhile replacing the naive error recovery used by TCP within the kernel by a more sophisticated scheme running in user space, we will see later that there are some drawbacks to doing this.

Although the peak bandwidth of TCP/IP is greater than UDP/IP in the graphs, BSPlib/UDP can achieve higher bandwidth utilisation (not shown here), although at the expense of a later rising curve.

The observed software latency at the *BSPlib* layer for 1400 bytes of user data is $11.6\mu s$ for packet download. This includes up to $7.8\mu s$ spent in copying the application data structure into a user/kernel space buffer with `memcpy()`. For packet upload (excluding memory copy into the application), the time for invoking the interrupt handler, doing any necessary error recovery, and uploading the packet is, on average, $3.6\mu s$ per packet. Due to the lightweight nature of this protocol, the observed efficiency on the wire is 93.62% (i.e., 93.62 *Mbps*) for the packet including our protocol headers (i.e., 1436 bytes).

Spray latency: Non-blocking communications can easily fail or suffer from deadlock, if two processes simultaneously push data into their protocol stacks in an attempt to send large amounts of data to each other. The point at which this deadlock occurs depends upon the buffering capacities of both sender and receiver. Quoting from the MPI report: "*. . . the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in 'pathological cases'.*" [13]. In MPI, if the error code of the send is ignored and data continues to be sent by a process, deadlock soon occurs. Although well-defined MPI programs are supposed to check the error code, and users deserve everything they get if

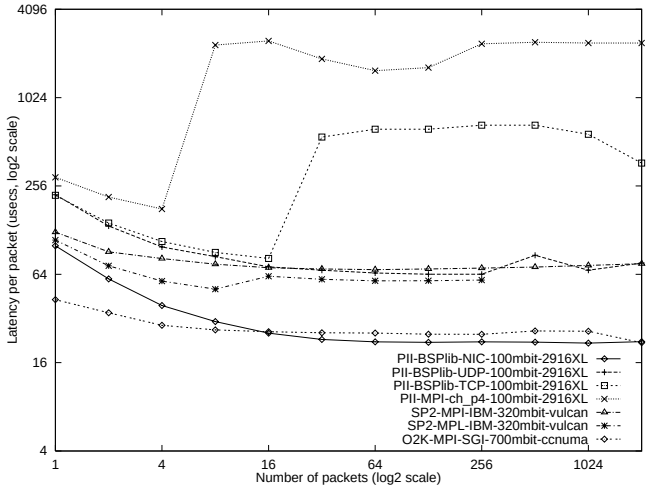

Fig. 3. Latency per packet (half roundtrip) for 4-byte messages as a function of number of packets.

they ignore it, the benchmark described in this section is intended to quantify the effectiveness of the buffering capacity of the various messaging systems by continually sending data in an attempt to exhaust buffers.

Figures 3 and 4 show the gap between successive send calls in the steady state, for 4 byte and 1400 byte messages. The benchmark combines the features of the round-trip and bandwidth benchmarks: a process sends i packets to a receiving process, which reflects them back. However, the application layer on the sender will not receive any of the reflected packets until all packets have been emitted, thus requiring them to be buffered in the protocol stacks or the network. For i packets shown on the horizontal axis, the time shown on the vertical axis is the time between sending the first packet and waiting for the last packet to return, *divided by* i. In the limit, this benchmark measures the latency between two successive send calls. An alternative interpretation, assuming deadlock does not occur, is that it measures the level of pipelining of communication, i.e., at $i = 1$ it is equivalent to the round-trip benchmark; for $i > 1$ it measures the degree of communication overlap due to packets in flight.

Before describing the results of this benchmark, it should be noted that the BSPlib/UDP and BSPlib/NIC messaging layers are used by *BSPlib* in such a way *that this form of deadlock cannot occur*. This is achieved by ensuring that if the protocol stack is unable to accept a send operation, then *BSPlib* will attempt to either: (1) consume a packet if one is available; or (2) request an acknowledgement to be returned on those links that have a large number of unacknowledged send buffers. In BSPlib/TCP, deadlock is avoided by using global flow control via slotting [10] as well as behaving as in point (1) above.

All the figures show that, regardless of packet size, for a large number of messages in flight (i.e., bulk communication as in BSP) BSPlib/NIC outperforms all other configurations, including the SP2 and O2K. All MPI implementations on the Cluster, SP2, and O2K suffer from deadlock when more than 256 packets

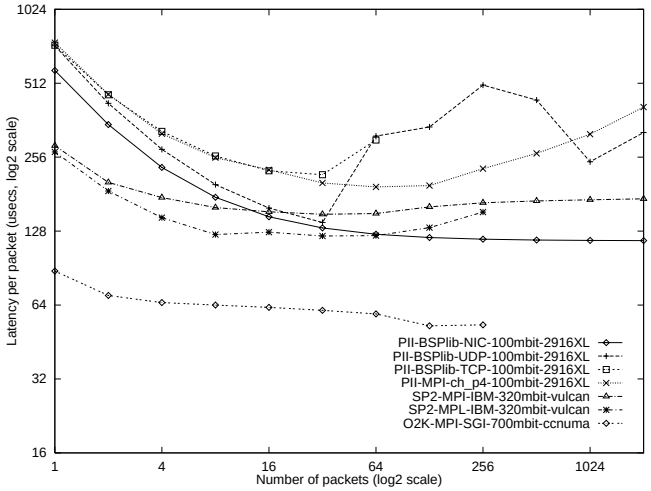

Fig. 4. Latency per packet (half roundtrip) for 1400-byte (approximately full-frame) messages as a function of number of packets

	p	SP2		Cluster					Origin 2K
		MPI	*BSPlib*	MPI	*BSPlib*				MPI
		IBM	MPL	`mpich`	TCP	UDP-1	UDP-2	NIC	SGI
BT	4	31.20	36.44	51.57	50.96	36.52	56.07	56.18	51.10
SP	4	24.89	27.31	33.15	40.86	18.37	41.85	42.36	56.22
MG	8	36.33	36.78	21.02	36.05	31.11	38.25	39.62	36.16
LU	8	38.18	36.50	46.34	55.50	50.15	54.63	63.09	87.34

Table 1. Results in Megaflops/sec *per process* for class-A NAS parallel benchmarks

are injected into the protocol stack. Although the roundtrip time on the cluster is $95\mu s$ for a single 4-byte packet, multiple packets can be injected into the protocol stack every $21\mu s$. While it is possible to do so faster ($8.2\mu s$ is the minimum), the protocol will not settle to a steady state, as the receiver cannot keep up with incoming packets, and overruns start to occur. Therefore $21\mu s$ was achieved by pacing the sender.

Figure 3 shows that the TCP/IP implementations (both BSPlib/TCP and `mpich`) all have a poor spray latency for large numbers of packets injected into the protocol stack. This is a consequence of the windowing flow control used by TCP where, if communication is run in a state where the window is always exhausted, throughput is considerably reduced. Again, this may lead one to conclude that it is worthwhile replacing the kernel-level TCP error-recovery mechanism, with a more sophisticated scheme in user space.

3.3 NAS Parallel Benchmark Performance

A BSP implementation [17] of version 2.1 of the NAS parallel benchmarks was used to compare the performance of BSPlib/TCP, BSPlib/UDP, and BSPlib/NIC on four and eight processors. Also, a standard MPI implementation of the bench-

	p	computation time	BSPlib/NIC	communication time and slowdown compared to BSPlib/NIC							
				`mpich`		BSPlib/TCP		BSPlib/UDP-1		BSPlib/UDP-2	
BT	4	$724.6s$	$24.2s$	$91.1s$	3.8	$100.9s$	4.2	$427.5s$	17.7	25.79	1.1
SP	4	$458.9s$	$42.8s$	$182.2s$	4.3	$61.2s$	1.4	$697.8s$	16.3	48.96	1.1
MG	8	$11.1s$	$1.1s$	$4.6s$	4.2	$2.4s$	2.2	$4.5s$	4.1	1.62	1.5
LU	8	$205.9s$	$30.5s$	$115.9s$	3.8	$62.8s$	2.1	$91.4s$	3.0	103.51	3.4

Table 2. Slowdown factors comparing each protocol BSPlib/NIC on the cluster

marks was used to compare the relative performance of the cluster with an 8-processor thin-node 66Mhz SP2, and a 72-processor 195Mhz O2K. Class-A-sized benchmarks were used. Due to memory limitations of the cluster, the FT benchmark was not performed.

Table 1 summarises the results of the NAS benchmarks for the three *BSPlib* protocols running on the cluster, comparing MPI variants of the benchmarks running on the cluster, SP2, and O2K. BSPlib/NIC outperforms `mpich` and IBM's proprietary implementation of MPI on the SP2 for all the benchmarks. On the O2K, BSPlib/NIC performs better on half of the benchmarks.

As the NAS benchmarks are mostly compute bound, large improvements in communication performance will only be realised as small improvements in the total Mflop/s rating of each of the benchmarks. As we aim to show the improvements of our communication protocol on the cluster compared to MPI, Table 2 gives a breakdown of time spent in computation and communication for each of the protocols running on the cluster. As all the benchmarks on the cluster use the same Fortran compiler (Absoft) and compiler options, and the computational part of the NAS benchmarks were not changed in any of the benchmarks, we assume the computation time spent in all the benchmarks is independent of the protocol (i.e., MPI, BSPlib/NIC, BSPlib/UDP, and BSPlib/TCP) used for communication.

Table 2 shows that BSPlib/NIC produces communication that is approximately 4 times better than `mpich` on all the NAS parallel benchmarks on the cluster. The improvements of the NIC protocol compared to BSPlib/TCP are not as marked as the `mpich` results, as some of the improvements are due to global optimisations that are applied in all implementations of *BSPlib* (i.e., packing and scheduling) [16]. As the NAS benchmarks communicate large amounts of data, the latency improvements of BSPlib/NIC over BSPlib/TCP will not be exercised to the fullest, and this accounts only for the approximately two times speedup in communication for these benchmarks.

Although BSPlib/UDP performs better than BSPlib/TCP in the micro benchmarks (see Section 3.2), the results given in the UDP-1 column of Table 1 show extremely poor performance in the computation bound benchmarks SP and BT. The problem with BSPlib/UDP compared to BSPlib/TCP (in which the error recovery is performed within the kernel by the TCP component of the protocol stack, and flow control is used) is that, in order for a message to be received asynchronously, a signal has to be dispatched to the user process and the user process has to be scheduled. To receive the message, the user process then makes

		Data rcvd	Explicit Acks	Data dropped	Duplicate rcvd
BT	UDP-1	893922	7.32%	38964	14913
$p = 4$	UDP-2	817471	0.42%	3490	0
	NIC	820233	2.53%	1	0
SP	UDP-1	1561491	6.58%	69076	21882
$p = 4$	UDP-2	1440148	0.18%	2534	0
	NIC	1422927	1.00%	14	0
MG	UDP-1	148023	4.21%	2339	1559
$p = 8$	UDP-2	141193	0.66%	926	0
	NIC	140113	1.51%	4	0
LU	UDP-1	1968612	2.99%	33681	3819
$p = 8$	UDP-2	1955673	2.14%	41878	7655
	NIC	2026933	6.23%	93	1900

Table 3. Packet statistics for the NAS parallel benchmarks

a `recvfrom()` system call. All this is on top of the height of the UDP/IP protocol stack. A number of effects come into play as a result of this extra path length and the delay in invoking recovery which BSPlib/NIC does not suffer from and which motivates our preference for a kernel level protocol: (1) The unreliable portion of the communiction is not inside the processor. This means that holding on to buffers and performing recovery at a level which includes a reliable path which also takes some time wastes buffer space. As the time between a message arriving at the NIC and its being uploaded into the user application may be large in BSPlib/UDP, more buffers will be required both for the buffering of incomming data and for unacknowledged data in the send queues (a direct result of Little's Law) there is a potential for packet loss due to overruns. (2) Both BSPlib/NIC and BSPlib/UDP reserve the same amount of buffer resources for communication. However, the NIC implementation, which also has a quicker acting interrupt handler, can carefully control their use, whereas BSPlib/UDP can only associate buffers with a point-to-point link by setting an appropriate socket option when the link is created. (3) By processing the protocol is the user space, the number of schedules/dispatches to process the user workload is increased slightly as some of the quanta is consumed processing the protocol. It is even concievable that the extra non-locality in the executable code may slow some processors by introducing TLB and cache stalls (but these effects have not been measured).

Table 3 shows the number of dropped and duplicate packets occurring under the NIC and UDP/IP implementation of *BSPlib* (remember that both protocols use exactly the same error recovery and acknowledgement policy). The marked difference between the error rate of the two protocols stems from the judicious use of buffering resources, which inhibits overruns, in BSPlib/NIC. On close investigation of the BT and SP benchmarks it was found that most of the packets were dropped by only one of the communication calls in each of the programs, yet these calls communicated the largest amount of data, and were performed regularly within each benchmark. Therefore although the error rate for the complete benchmark was only 4.3% for BT and 4.4% for SP, the actual error rate in the problematic section of communication was 99.2% and 81.1% respectively.

On investigation, as the Cisco switch did not drop any packets during any of the benchmarks, it was concluded that the high error rate was caused by

overruns occurring due to insufficeint buffer capacity within the IP part of the protocol stack in BSPlib/UDP. The hypothesis was tested by performing a crude form of flow control, whereby the sender was *throttled* so that packets could not be injected into the protocol stack any faster than $120\mu s$ (results are presented in the tables under the heading UDP-2). As can be seen from the UDP-2 data in Table 3, not only does this technique drastically reduce the number of dropped packets, but it reduces the number of redundant messages. This is due to BSPlib/UDP suffering from the problem of stale error recovery information being contained within packet headers due to a large number of packets being queued up for communication by the NIC[5]. Unfortunately, the limitation of the throttling is that it considerably lengthens the round-trip time for packets, although, as can be seen from the UDP-2 column of table 1, it has the desired effect of improving the performance of the NAS benchmarks. From these result we conclude that as slowing down the sender enables the BSPlib/UDP implementation of the NAS benchmarks to produce similar results to the low-latency BSPlib/NIC implementation, the NAS parallel benchmarks are not latency bound.

4 Performance comparison with other NIC protocols

There exist protocols whose hardware requirements are as modest as ours, whose bandwidth is as high as ours, and whose reliability is as great, but there are very few that achieve all three simultaneously. However, all three aspects are critical to building clusters that scale and can be used to execute applications.

The early approaches to low-latency and high-bandwidth communication recognised the redundancy in the protocol stack and hence the necessity of simplifying it, eliminating buffer copies by integration of kernel-space with user-space buffer management, and collapsing a number of network layers. This is the approach used in the protocol of Brustolini [5], based on ATM, which achieves reliable low-latency, high-bandwidth performance close to that of the hardware. However, in that implementation the overhead of sending must be quite high as the sending process blocks until the message is placed on the network.

An approach that bypasses the need for buffer management is the Active Messages of von Eicken *et al.* [28]. In the active message scheme, messages contain the address of a routine in a receiving process which handles the message by sending a response and/or populating the data structures of the computation with the data in the payload. A prototype implementation for a SparcStation over ATM, in which message delivery is reliable, is described by von Eicken *et al.* [27]. Because the message handler routine runs in the user's address space, there has to be some mechanism to make sure that the receiving process is scheduled when a message arrives. In general, this requires some kernel changes. The prototype for the SparcStation solves this scheduling problem by having the receiving process poll for messages. Although active messages are intended to

[5] Remember that the BSPlib/NIC alleviates this problem by using a gathered send. This allows low-latency packet download to prepare more than 75 packets for transmission in the time for a single full frame to be communicated between two machines.

Group	Name	Network	μs	Mbps	Reliable	Ref.
Genova	GAMMA (Active Messages, polling)	100Mbps hub	13	98	×	[7]
Genova	GAMMA (Active Messages, IRQ)	100Mbps hub	17	98	×	[7]
Oxford	PII-BSPlib-NIC-100mbit-wire	100Mbps wire	29	91	√	
Cornell	U-Net/FE (P133, hub)	100Mbps hub	30	97	×	[29]
Cornell	U-Net/FE (P133, switch)	100Mbps switch	40	97	×	[29]
Oxford	PII-BSPlib-NIC-100mbit-2916XL	100Mbps switch	46	91	√	
Oxford	PII-BSPlib-TCP-100mbit-2916XL	100Mbps switch	103	70	√	
Oxford	PII-BSPlib-UDP-100mbit-2916XL	100Mbps switch	105	79	√	
ANL	PII-MPI-ch_p4-100mbit-2916XL	100Mbps switch	147	78	√	
SUNY-SB	Pupa	100Mbps switch	198	62	√	[26]
Utah	Sender Based Protocols	FDDI ring	22	82	√	[24]
	FDDI (theoretical peak)	FDDI ring	510	100	√	
NASA	MPI/davinci cluster FDDI	FDDI ring	818	73	√	[19]
Cornell	Active Messages (write, SS20)	ATM switch	22	44	√	[27]
Cornell	Active Messages (read, SS20)	ATM switch	32	44	√	[27]
Cornell	U-Net/ATM (P133)	ATM switch	42	120	×	[29]
CMU	Simple Protocol Processing	ATM switch	75	48	√	[5]
NASA	MPI/davinci cluster Fore ATM	ATM switch	1005	98	?	[19]
NASA	MPI/davinci cluster SGI ATM	ATM switch	1262	85	?	[19]
Lyon	BIP	Myrinet	4	1008	×	[20]
Princeton	VMMC (Myrinet 2ndG, P166)	Myrinet	10	867	?	[11]
UCB	VIA	Myrinet	25	230	√	[6]
Princeton	VMMC (SHRIMP)	Intel Paragon	5	184	?	[12]
SGI	O2K-MPI-SGI-700mbit-ccnuma	CC-Numa	17	325	√	
Kalsruhe	PULC (port-M PVM)	ParaStation	27	92	√	[3]
IBM	SP2 (theoretical peak)	Vulcan switch	40	320	√	
IBM	SP2-MPL-IBM-320mbit-vulcan	Vulcan switch	55	173	√	
IBM	SP2-MPI-IBM-320mbit-vulcan	Vulcan switch	67	173	√	
	Ethernet (theoretical peak)	10Mbps hub	465	10	√	
NASA	MPI/davinci cluster Ethernet	10Mbps hub	900	8	?	[19]
NASA	MPI/davinci cluster HIPPI	HIPPI	851	265	√	[19]

Table 4. Half roundtrip latency (μs) and Link Bandwidth (Mbps)

be one-sided, this polling activity is a compromise between two-sided and one-sided communication. This is not really an application programming issue as the intention is that the interface be used within a library or in code generated by a parallel-language compiler. BSP communication is also one-sided. Apart from the obvious programming benefits of not having to match send and receive requests, as long as a reasonable number of buffers are deployed, there is no necessity to schedule the distributed processes involved in a computation in synchrony as message disposal is detached from message receipt in BSPlib/NIC. From the results in this paper, there appears to be no detrimental effect from this loose coordination of the processes, as very high percentage bandwidths are being realised. We attribute this to the number of buffers made available, the fact that data can be packed to fill buffers, and the fact that errors are detected early with recovery not relying on process scheduling.

Active Messages have also been used at various other levels in implementing transport mechanisms: GAMMA [7] is an implementation of Active Messages over Fast Ethernet and achieves an impressive $12.7\mu s$ one-way latency and a full frame bandwidth of approximately 85% of the medium. This bandwidth rises to an asymptotic bandwidth of 98% of the medium for two processors sending frames back-to-back. However, the assumption is that the medium is reliable and that frames arrive in order.

Table 4 provides a summary of the performances of various research groups'

low-latency and high-bandwidth clusters. The U/Net protocol [29] has similar properties to BSPlib/NIC as they both use Fast Ethernet networks, and use similar technique for allocating shared user-kernel memory buffers. As can be seen from the results, the bandwidth and latency of the two are similar. Pupa [26] is also a low-latency communication system over Fast Ethernet. It concentrates on buffer management and provides a reliable transport protocol but does not make assumptions about the network except that packets arrive in order. A missing packet triggers the receiver to initiate recovery. Thus there is an implicit assumption that there is only one route between a pair of nodes. Pupa achieves a one-way latency of $198\mu s$ and achieves a bandwidth of 62Mbps for very large messages (10000 bytes) but achieves less than 60% efficiency for full frame-sized messages.

There are a number of protocols operating over Myrinet [4], for example BIP [20]. BIP (Basic Interface for Parallelism) has similar design objectives to Pupa: to support a message-passing parallel-computing environment (there is an MPI implementation for BIP). No error recovery is supported, but each packet receives a sequence number and contains a checksum, and hence errors can be detected. BIP achieves a good one-way latency of $4.3\mu s$ and an asymptotic bandwidth of 1008Mbps (96% of the available 1056Mbps).

The suitability of some of the network media mentioned in Table 4 for high-performance parallel computing may not be immediately obvious. Certainly, a single wire between two processors is not at all scalable and cannot be taken seriously (we only include such configurations in our our work to illustrate the contribution of the switch to the latency). Also, all of the bus-based schemes are not scalable; this includes the Fast Ethernet implementations that use an Ethernet hub, since the hub merely acts as an an extension of the bus and its bandwidth is divided amongst the communicating processors. Hub implementations imply the use of the CSMA/CD protocol, which makes no promise for reliable delivery, and hence this must be built into some higher protocol. For scalability purposes, the FDDI ring can also be considered a bus because the bandwidth is diluted amongst the communication processes. However, a single FDDI ring can be used in such a way that the higher-level protocols do not have to provide recovery.

Treating error recovery as optional based on the underlying reliability of a point-to-point benchmark is a false argument. Unless some back-pressure notification is made available to the sender, it will always be possible to overwhelm a component of the switch in the presence of unstructured communication patterns. Switches are starting to appear which address this problem, either by augmenting the protocol to send a flow-control packet back along the link, or by holding the cable busy so that the NIC CSMA/CD engine will hold off transmitting the next packet. However, this does not solve edge problems when multiple links are used.

Communication primitives in which the receiver polls on a device, for example in variants of active messages such as GAMMA [7] and U/NET [27], are not one-sided and the measurement of latency may hide a potentially-large loss in

CPU cycles. This loss can arise as a result of the sender running ahead of the receiver. If the send is truly synchronous, then the sender waits for the corresponding receive, else if the sender continues, it may re-invoke a send later and cause an overrun (or an overrun due to another processor sending data to the same destination). If the receiver runs ahead, then time will be wasted polling for the incoming message. Of course depending on the model, such wastage of CPU time may be inevitable, but it does not occur for well-designed BSP computations. Having tightly-coupled communication is difficult in the loosely-coupled environment of clusters; we have adopted a buffer-slackness approach to accommodate this loose coupling.

The switched solutions are the only ones that provide scalability (Myrinet, ATM and switched Fast Ethernet). Of the Fast Ethernet solutions, our results are best in terms of scalability, reliability, latency and bandwidth, and we are competitive with the much more expensive Myrinet solutions.

5 Conclusions

Many low-latency, high-bandwidth protocols for clusters have been developed, and their performance is generally impressive. However, it is not clear that this performance can be exploited in applications, primarily because providing reliability requires adding significant extra capabilities to a protocol.

The important question is: where should this extra functionality be added? The trend in high-performance clusters has been to increase the amount of protocol work that takes place in user space. Our results suggest that this may not be a good thing, because it has a significant effect on computational performance. We show that it is possible to achieve latency and bandwidth comparable to existing protocols while executing in kernel space. The important effect of this is that reliability can be achieved without significant performance degradation.

We document the performance behaviour of this new protocol stack, using low-level benchmarks to explore single-link behaviour, and the NAS parallel benchmarks to demonstrate that the single-link performance scales to applications.

Acknowledgements: Jonathan Hill's work was supported in part by the EPSRC Portable Software Tools for Parallel Architectures Initiative. David Skillicorn is supported in part by the Natural Science and Engineering Research Council of Canada. The authors would like to thank Oxford Supercomputing Centre for providing access to an Origin 2000.

References

1. 3Com. *3C90x Network Interface Cards Technical Reference*, December 1997. Part Number:09-1163000.
2. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yellow. The NAS parallel benchmarks 2.0. Report NAS-95-020, NASA, December 1995.
3. J. M. Blum, T. M.Warschko, and W. F. Tichy. PULC: Parastation user-level communication. Design and Overview. In *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 498–509. Springer, 1998.
4. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet — a gigabit-per-second local-area network. `http://www.myri.com`, November 1994.

5. J. C. Brustolini and B. N. Bershad. Simple protocol processing for high-bandwidth low-latency networking. Technical Report CMU-CS-93-132, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1992.
6. P. Buonadonna, A. Geweke, and D. E. Culler. Implementation and analysis of the Virtual Interface Architecture. In *SuperComputing'98*, 1998.
7. G. Ciacco. Optimal communication performance on Fast Ethernet with GAMMA. In *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 534–548. Springer, 1998.
8. Cisco Systems. *Catalyst 2900 series XL installation and configuration guide*, 1997. Part Number: 78-4417-01.
9. S. R. Donaldson, J. M. D. Hill, and D. B. Skillicorn. BSP clusters: high performance, reliable and very low cost. Technical Report PRG-TR-5-98, Programming Research Group, Oxford University Computing Laboratory, September 1998.
10. S. R. Donaldson, J. M. D. Hill, and D. B. Skillicorn. Predictable communication on unpredictable networks: Implementing BSP over TCP/IP. In *EuroPar'98*, LNCS, Southampton, UK, September 1998. Springer-Verlag.
11. C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 388–396. IEEE, IEEE Press, 1997.
12. C. Dubnicki, L. Iftode, E. W. Felton, and K. li. Software support for virtual memory mapped communication. In *Proceedings of the 10th International Parallel Processing Symposium*. IEEE, IEEE Press, 1996.
13. M. P. I. Forum. *MPI A Message-Passing Interface Standard*, May 1994.
14. W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.
15. J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, November 1998. see `www.bsp-worldwide.org` for more details.
16. J. M. D. Hill and D. Skillicorn. Lessons learned from implementing BSP. *Journal of Future Generation Computer Systems*, 13(4–5):327–335, April 1998.
17. A. L. Hyaric. Converting the NAS benchmarks from MPI to BSP. Technical report, Oxford university Computing laboratory, 1997. Available from `ftp://ftp.comlab.ox.ac.uk/pub/Packages/BSP/NASfromMPItoBSP.tar`
18. Mier Communucations Inc. Product lab testing comparison: 10/100BASET switches, April 1998.
19. National Aeronautics and Space Administration. Summary of recent network performance on davinci cluster. `http://science.nas.nasa.gov/Groups/LAN/cluster/latresults/-sumtab.recent.html`.
20. L. Prylli and B. Tourancheau. A new protocol designed for high performance networking on Myrinet. In *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 472–485. Springer, 1998.
21. A. Rubini. *Linux Device Drivers*. O'Reilly and Associates, 1998.
22. A. Simpson, J. M. D. Hill, and S. R. Donaldson. BSP in CSP: easy as ABC. Technical Report PRG-TR-6-98, Programming Research Group, Oxford University Computing Laboratory, September 1998.
23. D. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.
24. M. R. Swanson and L. B. Stoller. Low latency workstation cluster communications using sender-based protocols. Technical Report UUCS-96-001, Department of Computer Science, University of Utah, Salt Lake City, UT 84112, USA, 1996.
25. L. G. Valiant. Bulk-synchronous parallel computer. U.S. Patent No. 5083265, 1992.
26. M. Verma and T. cker Chiueh. Pupa: A low-latency communication system for Fast Ethernet, April 1998. Workshop on Personnel Computer Based Network of Workstations held at the 12th International Parallel Processing Symposium and the 9th Symposium on Parallel and Distributed Processing.
27. T. von Eicken, V. Avula, A. Basu, and V. Buch. Low-latency communication over ATM networks using Active Messages. Technical report, Department of Computer Science, Cornell University, Ithaca, NY 14850, 1995.
28. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, volume 20(2) of *ACM SIGARCH Computer Architecture News*. ACM Press, May 1992.
29. M. Welsh, A. Basu, and T. von Eicken. Low-latency communication over Fast Ethernet. In *EuroPar'96 Parallel Processing: Volume I*, volume 1123 of *Lecture Notes in Computer Science*, pages 187–194. Springer, 1996.