Exploiting Global Structure for Performance on Clusters

Stephen R. Donaldson
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, U.K. OX1 3QD
Stephen.Donaldson@comlab.ox.ac.uk

Jonathan M.D. Hill
Sychron Ltd.
1 Cambridge Terrace
Oxford, U.K. OX1 1UR
jon@sychron.com

David B. Skillicorn
Computing and Information Science
Queen's University, Kingston
Canada K7L 3N6
skill@qucis.queensu.ca

Abstract

Most parallel programming models for distributed-memory architectures are based on individual threads interacting via `send` *and* `receive` *operations. We show that a more structured model, BSP, gains substantial performance improvements by exploiting the extra information implicit in its structure. In particular, each thread learns something about global state whenever it receives a message. This information can be used to modify its own behavior to improve collective use of the communication system. The programming model's semantics also provides implicit knowledge that can be exploited to increase performance. We show that these effects are useful at the application level by comparing the performance of BSP and MPI implementations of the NAS parallel benchmarks.*

1. Introduction

Message-passing programming models for distributed-memory architectures have a natural vertical structure. Each thread is almost an independent program, except that it contains occasional `send`s and `receive`s. The ability to write programs 'one thread at a time' is natural, and so this vertical structure simplifies program design. However, it also makes efficient implementation difficult. It is pleasant for the programmer to be able to write a `send` anywhere in the thread, but it it is unpleasant for the thread to have to be ready to handle a `receive` at any moment. An implementation of such a model knows very little about what may happen next, and therefore must make the most conservative decisions at any point, ruling out many potential performance improvements.

In contrast, some programming models have a natural horizontal structure. Programs are sequential compositions of operations that are internally parallel and fill the whole target computer. Skeletons (also sometimes called templates) are one example; Bulk synchronous parallelism (BSP) [10] is another, and forms the focus of this paper.

In horizontal models, programmers are not free to insert communication actions anywhere in their code; instead, communications are constrained to take place only in certain places. However, and this is the crucial point, this restriction on communication means that each processor can infer the global state of the computation from the communication in which it is involved. This knowledge of the global state translates into performance enhancements. We show that, on a cluster, *BSPlib* can achieve communication performance improvements of a factor of approximately four over MPI, to which it is functionally similar. This improvement is largely attributable to the exploitation of explicit and implicit knowledge based on global structure.

Thread-based programming languages are naturally implemented using libraries, since they only require the addition of a few communication operations to an ordinary sequential language. BSP has also been implemented as a C and Fortran library, *BSPlib* [7].

We describe the design of *BSPlib* in three layers: the library itself, a messaging layer, and a transport layer, specialized either to use datagrams (UDP/IP) or network interface cards directly. Each of these layers is designed to work only in the context of *BSPlib*. Knowing the pattern of calls it will encounter (explicit global information) and that it is implementing supersteps (implicit global knowledge) are both critical to design for maximum performance. The paper continues a series [5, 8] showing how the global

structure of BSP can be exploited to design implementations with increased performance. BSP Implementations strive to improve the effectiveness with which each real parallel computer emulates a BSP abstract machine.

Section 2 introduces the BSP programming model and abstract machine, and the current most-popular implementation, *BSPlib*. Section 3 describes the messaging layer of the *BSPlib* implementation and three aspects of its design that rely on BSP superstep structure. Section 4 introduces the transport layer, describes how error recovery is handled, and how BSP structure is used to reduce acknowledgement traffic overheads. Section 5 presents performance data for the protocol stack, including measurements of application performance using the NAS parallel benchmarks. Section 6 discusses other low-latency protocols on clusters. Finally, we draw some conclusions in Section 7.

2. The BSP Model and *BSPlib*

BSP [10] is a model of parallel computation whose abstract machine is a collection of p processor-memory pairs, an interconnection network or switch through which the processors communicate packets in a point-to-point manner, and a mechanism by which the processors can barrier synchronize. The model assumes nothing about the topology or routing protocol of the interconnection substrate, but parameterizes the switch in terms of its permeability to continuous messages (g) and its global barrier synchronization latency (l). The g parameter is, in effect, the inverse of the effective end-to-end bandwidth. It often differs markedly from bandwidth figures provided by manufacturers since it measures how data transmission behaves under realistic conditions, when traffic is dense, and traffic patterns are unstructured. Any parallel computer can readily emulate the BSP abstract machine. However, BSP libraries play a significant role in improving the efficiency of this emulation (improving the effective values of l and g).

BSP programs are written in *supersteps* which are global operations of the entire machine. Each superstep consists, semantically, of three sequential phases: (1) a computation phase in which each processor computes with locally-held values, (2) a communication phase in which data moves among processing elements, and (3) a barrier synchronization. Data is not visible to user code at its destination until after the barrier synchronization ending the superstep in which it was transferred. *BSPlib* [7] is a small communication library of 20 primitives, used in an SPMD style.

3. Messaging layer design for *BSPlib*

The *BSP messaging layer* is responsible for providing an interface on which the upper layer *BSPlib* primitives can be implemented. The BSP messaging layer is similar to the Abstract Device Interface used in the `mpich` implementation of MPI. The bulk of the work in porting *BSPlib* amounts to rewriting the messaging layer, and providing a binding to the new transport layer (there is also a similar structure for shared memory implementations of *BSPlib*). The messaging layer is not a general interface, and in particular, takes into account how it is used by the upper layers of the *BSPlib* implementation. This restriction in the interaction of the two layers allows *BSPlib* to guarantee that the computation is both deadlock and livelock free. Indeed, this can be seen as a direct result of uncoupling the user one-sided communication requests from their implementation by *BSPlib* on the available transport layer.

The BSP messaging layer implements the semantics of supersteps (on distributed-memory systems) as follows:

1. Calls to `bsp_sync` begin the actual transfers of data implied by previous one-sided `put` calls in the superstep. Because all of the data sent by each processor is handled at the same time, *BSPlib* can use *repackaging* to build message units that best suit the underlying communication substrate.
2. Each processor creates a bit-matrix specifying the processors with which it will communicate. A reduction of these matrices with bitwise-or produces a matrix that contains all of the information about who will communicate with whom; and each processor can compute this result with the same time complexity using a variant of parallel prefix or scan. (Note that this reduction *is* the barrier of the superstep);
3. A total exchange takes place between those processors involved in communication in the superstep, so that each processor informs the others of the number and size of messages it plans to send to them;
4. Each processor sends its messages using *destination scheduling* to determine the order, and *pacing* to determine how fast to insert data into the network.
5. When each processor has received all the messages it expects, it continues to the computation phase of the next superstep.

Repackaging, *destination scheduling*, and *pacing* all rely on the semantics of the BSP model and the global knowledge accessible to each processor.

Repackaging depends crucially on the decision to postpone communication until the end of each superstep. This has an obvious performance penalty, the lost factor of (at most) two available by overlapping computation and communication. However, experiments have shown that this factor is recovered by better use of the interconnect. On many architectures, the performance gain from repackaging is significantly greater than a factor of two, especially if

the user application naturally requires small user messages [8].

Destination scheduling also depends on postponing communication until after computation. A total exchange in an SPMD program makes the following likely: each processor first tries to send to Processor 0, then to Processor 1, and so on. Each processor in turn is bombarded with many messages, can only extract the first one, and creates congestion back through the network. If processors all send their messages in different orders, there will be approximately one message arriving at each destination at all times. *BSPlib* achieves this in a variety of ways, e.g., randomizing destinations, or using a latin square as a transmission schedule, but the ability to do so depends critically on the structure of superstep communication [8].

Pacing obtains the maximum throughput from the communication system as a whole. A graph of throughput as a function of applied load for any interconnection network has a characteristic shape. As applied load increases, so does throughput, until some resource in the network saturates. After this point, throughput no longer increases, and usually decreases, often rapidly. A messaging layer should attempt to pace injection of new messages into the network so that the total applied load is just below the peak of this curve (just below, so that slight variations in load are met by slightly greater capacity, i.e. negative feedback).

This kind of pacing is not possible for most commercial parallel computers because it is extremely difficult to determine what the throughput versus load characteristics are, and because the software below the manufacturer's message-passing interface inserts delays that are unpredictable enough that the desired behavior cannot be achieved anyway. However, in networks of workstations and clusters, the level accessible to the messaging layer is close enough to the hardware that pacing becomes important, indeed essential when shared media are used.

In the BSP context, a processor that is about to communicate knows which other processors are about to communicate, and how much they plan to send, and therefore *can* know (approximately) the global applied load. Exploiting this knowledge both to determine an ideal slot size, and to schedule each processor's attempt to transmit has been shown to significantly improve the performance of TCP/IP on shared media such as Ethernet [5].

None of these techniques can be used by message-passing systems such as MPI or PVM, or even by LogP. Repackaging and destination scheduling are ruled out, except perhaps in some *ad hoc* way, by the lack of a definable moment to do them. Pacing is also not possible because each processor knows the load *it* is about to apply to the communication network but, in the absence of knowledge of the global applied load, can't tell if this is a good time to send because the network is lightly loaded, or a bad time because it is congested.

4. Transport layer design

We describe two implementations of the transport layer: (1) a BSPlib/UDP implementation that uses BSD datagram sockets as an interface to UDP/IP; and (2) a BSPlib/NIC implementation that uses the same error-recovery and acknowledgement protocol, but replaces the BSD datagram interface with a lightweight packet transmission mechanism that interfaces directly with network interface cards.

BSPlib/UDP is built on the BSD datagram socket interface, which provides unreliable, full-duplex, connectionless packet delivery between machines using UDP/IP. UDP/IP provides no form of message acknowledgement, error recovery, or flow control, so user APIs such as *BSPlib* must handle these themselves. In the protected kernel/user-space model of UNIX, this must be done in user space.

BSPlib/NIC is built upon a lightweight packet transmission mechanism that interfaces directly to Network Interface Cards (NICs). General-purpose NICs (e.g., Myrinet), are controlled by dedicated microprocessors that run user-supplied programs concurrently with the host microprocessor. At present, these NICs are produced in low volume and are relatively expensive. In contrast, high-volume fixed-protocol NICs have a fixed firmware program, whose parameters can be altered via a memory-mapped register file. Due to the price (typically $US100) and installed user base (millions of units) of such NICs, we use them in our cluster.

4.1. Basic protocol design

The protocols for BSPlib/UDP and BSPlib/NIC are identical, except in the precise interface to the lower level, and the location of error recovery. BSPlib/UDP implements a reliable send by copying the user data structure references in a send call to a buffer taken from a free queue. In BSPlib/NIC, the interface between the messaging layer on the NICs is achieved by having a portion of memory, used for communication buffers and shared data structures, mapped into both the kernel and the user's address space. The *BSPlib* layer enqueues an element directly onto the NIC's transmit queue (and also onto a send queue for recovery purposes). This is serviced asynchronously by the NIC, which will poll for work if none is available. To avoid protocol information becoming stale, the NIC performs a gathered send, where protocol information is only read when the packet is about to be transmitted.

4.2. Adding reliability

Many existing low-latency protocols for clusters do not address reliability. Their designers argue that the probabil-

ity of link errors is close to that of processor failures, and therefore there is no benefit to incorporating reliability. In practice, however, transmission failures do occur, not because of link failures but because of buffer overruns at destination processors and in switches themselves. Such occurrences cannot be distinguished from link errors. If low-latency protocols are to be used for practical computing, reliability must be an integral part of their design.

High-performance protocols must maximize the bandwidth available to user programs. Bandwidth is reduced whenever (a) packets must be retransmitted because of loss, and (b) control information must be sent to coordinate protocol activity at each node.

In clusters, errors most often occur because of buffer overruns at destination processors. As a result, errors tend to occur in bursts. Our transport protocol uses *hole filling* rather than *go-back-n* as its recovery strategy.

Two kinds of control information are required: requests for retransmission and acknowledgements. When traffic patterns are symmetric, acknowledgements are piggybacked onto traffic going in the opposite direction. When traffic is not symmetric, it is important to minimize the number of explicit extra packets. In the *BSPlib* transport layer, communication takes place immediately after a total exchange. Each processor knows how many packets it is expecting from every possible sender. This means that the decision to generate an explicit acknowledgement (at the expense of bandwidth) can be (receiver) resource-based rather than (sender) timeout-based.

4.3. Error recovery by hole-filling

Two types of packets are used in the protocol: (1) Sequenced data packets that contain a fixed-sized header with MAC, sequence number, payload length and type, and piggybacked recovery information. This is followed by a variable-length payload, whose length is bounded so that the entire packet is no larger than the physical layer's frame size; and (2) Unsequenced control packets that contain the same fixed-sized header, but no data. Control packets are used for explicit acknowledgements and *prods*, as described below. Each process contains a free queue of packets (in the BSPlib/NIC implementation these are mapped into kernel and user space), and send and receive queues associated with each communication channel ($p-1$ in total). The send queue contains unacknowledged sent data-packets, and the receive queue contains data-packets not yet consumed by the upper *BSPlib* layer.

Consider two processors, A and B, communicating over a channel that loosely guarantees packet ordering[1], but which may drop packets. If process A sends the sequence of packets $[0, 1, \ldots, 9]$ to B, and only packets $[0, 1, 4, 5, 6, 9]$ arrives at B, the recovery protocol must ensure that A learns of these missing packets.

In our hole-filling selective retransmit protocol, the fixed header contains two fields for piggybacked sequence number information: (1) An *acknowledgement* field (if B sent a message back to A after having received $[0, 1, 4, 5, 6, 9]$, this field would contain 1 allowing A to remove packets 0 and 1); and (2) An *end-of-hole* field which specifies the sequence number of the packet after the last in-sequence packet, i.e., 4. When no unaccounted-for packet loss has occurred, the acknowledged sequence number and this field are the same. If A and B are involved in symmetric communication, then A learns of the first hole from this piggybacked data and resend packets 2 and 3. After these have been received by B, further piggybacked information will trigger the resending of 7 and 8. To prevent potentially stale in-flight information from being re-used to cause repeated resends, a double retransmission of data is delayed for at least a round-trip delay of the network.

If there is no reverse traffic from B to A then, when the upper *BSPlib* layer has consumed packets $[0, 1]$ and now requires packet 2, B will *prod* process A with a control packet. This allows A to learn of the hole and resend packets 2 and 3. In case the control packet goes missing, this prodding is repeated no more frequently than the round trip interval of the network. This prodding can also arise without packet loss when the computation on a processor runs ahead of another and hence enters its communication phase earlier. In this case the prodded processor responds with an acknowledgement rather than missing data. The prodding processor then realizes that it has run ahead and is being impatient, and enters an exponential back-off of prods.

The benefits of this scheme over a more-general selective retransmission scheme is that only one integer field is required in the packet header. The network used, and the scheme we employ, ensures that packet loss is quite rare (0.05% packet drops are experienced in the NAS benchmarks in Section 5) and any packets lost are actually dropped by the receivers when, for example, available buffers run low. When buffers run low, only the next expected packets are accepted, but the acknowledgement data on all arriving packets is still inspected so that buffers held in the send queues can be freed, and progress is guaranteed.

4.4. Minimizing non-piggybacked acknowledgements

We use two techniques, one fine-grained, the other coarse-grained, for handling the acknowledgement of messages when there is no reverse traffic on which to piggyback them. The overall strategy is to try as hard as possible to avoid sending acknowledgements. This involves waiting

[1] Our cluster guarantees ordering, but the software can handle out-of-order arrivals within a configurable tolerance.

System		Latin sq order	pid order	Factor
SP2	big	102547	107073	1.0
T3D	big	14658	29017	2.0
Pwrchallenge	big	37820	47910	1.3
Cluster(TCP)	small	61336	119055	1.9
	big	134839	248724	1.8
Cluster(NIC)	small	39468	76597	1.9
	big	78683	14043686	178

Table 1. Time for an 8-processor total exchange (μs) with and without destination scheduling; small=16k words per processor pair, big=32k words.

Msg Size	mpich	BSPlib/NIC	Improve
	Roundtrip (μs)		
4	297	93	3.2
256	348	170	2.1
1024	635	348	1.8
1400	754	471	1.6
	Bandwidth (Mbps)		
4	0.006867	1.458829	212.4
256	0.439494	46.626926	106.1
1024	53.436917	89.611876	1.676
1400	77.779662	91.271637	1.173

Table 2. Latency and bandwidth as a function of message size.

much longer than a conventional implementation would before sending explicit acknowledgements.

The coarse-grained acknowledgement scheme utilizes the superstep structure of *BSPlib* programs, and acknowledges data implicitly. Suppose that processor i has sent unacknowledged data to processor j in a previous superstep. If processor j is below processor i in the reduction tree that is used to determine which processors plan to communicate, or if processor j has data to send to processor i, then processor i will receive a control message from j onto which an acknowledgement can be piggybacked at no cost. However, it may be that neither of these possibilities happens. The need for an explicit acknowledgement can still be avoided. If processor i receives any message (from any processor) that comes from the next superstep, it can safely assume that all processors (including j) have logically passed the most recent barrier, and hence can treat the sent messages as acknowledged. Knowing which superstep a message comes from is achieved by marking when barriers took place on message queues (logically, using alternating colors to label messages from successive supersteps).

The fine-grained scheme is based upon volume of communication and has both a sender and receiver component. 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 anticipates when buffer resources will run out, and attempts to force an acknowledgement to be returned just before buffer starvation. This minimizes 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 was requested, and at least $\frac{n}{2p}$ elements have come in over a link since the last acknowledgement (either piggybacked or explicit).

The fine-grained scheme can be refined further by using information from the upper *BSPlib* level. At the end of each communication phase (superstep), the protocol also marks the last data item sent to a process as requiring an explicit acknowledgement. This improves the usefulness of acknowledgements. Such last transmissions are likely to result in acknowledgements anyway, and explicitly requesting them gets them sent earlier and at a time when traffic on the interconnect is quiescing. Also, this increases the number of free buffers at the start of the next communication phase and makes non-piggybacked acknowledgements less likely during that communication phase.

5. Performance improvements

We compare the performance of *BSPlib* with MPI because both libraries have about the same functionality; indeed many programs can globally substitute one set of calls for the other and execute without further modification. MPI has the vertical structure of independent threads connected by messages, while *BSPlib* has a strong horizontal structure. Thus performance differences can be attributed to differences in design made possible because of additional assumptions. The cluster used for benchmarking consists of eight 400MHz Pentium II PC systems, with 128MB of 10ns SDRAM on 100MHz motherboards, connected by 100 Mbps switched Ethernet. The same Fortran compiler (Absoft) and compiler options were used for all programs, and the computational part of the NAS benchmarks was not altered.

At the application level, we compare a BSP implementation of version 2.1 of the NAS parallel benchmarks [1] using BSPlib/UDP and BSPlib/NIC implementations on the cluster, `mpich` on the same cluster, and standard MPI implementations of the benchmarks on an 8-processor thin-node 66Mhz IBM SP2, and a 72-processor 195Mhz Origin 2000. Class-A-sized benchmarks were used. Due to memory limitations of the cluster, results from the FT benchmark are not given. The four benchmarks are: BT, an application that solves systems of blocked-tridiagonal linear equations; SP, an application that solves systems of scalar-pentadiagonal linear equations; MG, a multigrid benchmark; and LU, which solves a system of linear equations using LU decomposition. LU performs many small (5 word)

	p	SP2		Cluster				Origin 2000
		MPI	*BSPlib*	MPI	*BSPlib*			MPI
		IBM	MPL	mpich	TCP	UDP	NIC	SGI
BT	4	31.20	36.44	51.57	50.96	56.07	56.18	51.10
SP	4	24.89	27.31	33.15	40.86	41.85	42.36	56.22
MG	8	36.33	36.78	21.02	36.05	38.25	39.62	36.16
LU	8	38.18	36.50	46.34	55.50	54.63	63.09	87.34

Table 3. Results in Megaflops/sec *per process* **for NAS parallel benchmarks (class A) v2.1**

communications, and therefore provides a good measure of the communication latency of a system.

Repackaging can be significant for the programs that communicate using many small messages, but does not have much effect for application programs whose communication demands are large. The NAS benchmarks are certainly in the latter class.

Destination scheduling improves performance by a factor of about two for a wide range of parallel computers. Some typical performance data are shown in Table 1. Note that the use of a latin square delivery schedule consistently reduces delivery time by a factor of about two but, for the cluster using BSPlib/NIC, it has the added side-effect of avoiding the nasty performance surprise that happens when buffers in the switch saturate.

It is hard to measure the effect of pacing injection of new data into the network and the use of an appropriate slot size. For all of the BSP implementations, pacing and slotting are essential to make the behavior of communication predictable. However, improvements over mpich of more than a factor of two have been reported for TCP/IP, and factors of up to 1.4 for UDP/IP [5].

The low-level performance of our transport protocol is described in [6]. Table 2 gives some example measurements. The performance improvement due to the transport protocol depends heavily on the size of messages transmitted. For full-frame packets, the effective improvement in performance comes both from the $\sim$60% improvement in latency, and the $\sim$17% improvement in bandwidth.

To ensure that these low-level performance improvements translate into increased performance for application programs, we present performance results for the NAS benchmarks (Table 3). The BSPlib/NIC implementation outperforms mpich and IBM's proprietary implementation of MPI on the IBM SP2 for all the benchmarks. On the Origin 2000, the BSPlib/NIC implementation performs better on half of the benchmarks.

As the NAS benchmarks are mostly compute-bound, large improvements in communication performance only produce small improvements in the total Mflop/s rating of each of the benchmarks. Table 4 gives a breakdown of time spent in computation and communication for each of the protocols running on the cluster. The communication time is calculated using a BSP trace profiling tool. We assume the computation time spent in all the benchmarks is independent of the communication protocol. The results for the BSPlib/UDP implementation appear poor because error recovery takes place in user space, which increases the computation time, and this is not taken into account in Table 4 (although its true performance is apparent in Table 3). Table 4 shows that the BSPlib/NIC implementation of *BSPlib* produces communication that is approximately 4 times faster than mpich for all the NAS parallel benchmarks on the cluster.

	p	comp time	comm time and slowdown compared to BSPlib/NIC				
			NIC	mpich		UDP	
BT	4	$724.6s$	$24.2s$	$91.1s$	3.8	$25.8s$	1.1
SP	4	$458.9s$	$42.8s$	$182.2s$	4.3	$49.0s$	1.1
MG	8	$11.1s$	$1.1s$	$4.6s$	4.2	$1.6s$	1.5
LU	8	$205.9s$	$30.5s$	$115.9s$	3.8	$103.5s$	3.4

Table 4. Slowdowns relative to the BSPlib/NIC **protocol on the cluster**

		Recvd	ExplAck	Drops	Dups	ImplAck
BT	UDP	817471	0.42%	3490	0	0.7%
$p = 4$	NIC	820233	2.53%	1	0	0.3%
SP	UDP	1440148	0.18%	2534	0	1.0%
$p = 4$	NIC	1422927	1.00%	14	0	0.4%
MG	UDP	141193	0.66%	926	0	8.1%
$p = 8$	NIC	140113	1.51%	4	0	6.8%
LU	UDP	1955673	2.14%	41878	7655	32.0%
$p = 8$	NIC	2026933	6.23%	93	1900	30.5%

Table 5. Packet statistics for the NAS parallel benchmarks

Table 5 compares the number of extra error-recovery packets generated in the BSPlib/UDP and BSPlib/NIC implementations of the same protocol. The high number of duplicates and data dropped in the UDP/IP implementation is because packets are being dropped by the kernel, somewhere between their reception at the processor, and the SIGIO signal being scheduled to the user process. This highlights the importance of slackness in the buffer pool *at the lowest level of the protocol*. The column labelled "implicit ack" identifies the percentage of packets that were reclaimed from the send queue due to the coarse-grained acknowledgement scheme. In most of the NAS benchmarks, piggybacked acknowledgements and explicit fine-grained acknowledgements are used to remove most of the packets from the send queue. Only in LU is there enough communication to exploit the coarse-grained scheme. This results in the highest number of explicit acknowledgements generated by all the benchmarks, as well as 30.5% of all packets sent being acknowledged using the coarse-grained acknowledgement scheme of Section 4.4.

6. Related Work

Very little performance data exists for many cluster protocols other than low-level latency and bandwidth measurements. To give some picture of how our results compare to

Group	Name	Network	μs	Mbps	Ref.
Oxford	PII-BSPlib-NIC-100mb-2916XL	100Mbps switch	46	91	
Oxford	PII-BSPlib-TCP-100mb-2916XL	100Mbps switch	103	70	
Oxford	PII-BSPlib-UDP-100mb-2916XL	100Mbps switch	105	79	
ANL	PII-MPI-ch_p4-100mb-2916XL	100Mbps switch	147	78	
SUNY-SB	Pupa	100Mbps switch	198	62	[12]
Utah	Sender Based Protocols	FDDI ring	22	82	[11]
	FDDI (theoretical peak)	FDDI ring	510	100	
NASA	MPI/davinci cluster FDDI	FDDI ring	818	73	
Cornell	Active Messages (write, SS20)	ATM switch	22	44	[13]
Cornell	Active Messages (read, SS20)	ATM switch	32	44	[13]
CMU	Simple Protocol Processing	ATM switch	75	48	[2]
UCB	VIA	Myrinet	25	230	[3]
SGI	O2K-MPI-SGI-700mbit-ccnuma	CC-Numa	17	325	
IBM	SP2-MPL-IBM-320mbit-vulcan	Vulcan switch	55	173	
IBM	SP2-MPI-IBM-320mbit-vulcan	Vulcan switch	67	173	
NASA	MPI/davinci cluster HIPPI	HIPPI	851	265	

Table 6. Half roundtrip latency (μs) and Link Bandwidth (Mbps). Rows without citations are our measurements.

this other work, Table 6 summarizes data for other protocols that are (a) scalable as the number of processors grows, and (b) reliable. This omits some of the highest-performance protocols, for example Gamma [4], Unet [14], and BIP [9] which do not address reliability (and hence do not have to handle recovery). All of the protocols in the table claim to provide reliability but it is not always obvious how much this was tested in the performance measurements reported.

7. Conclusions

We describe the design and performance of a protocol stack designed for a programming model that has horizontal structure. This structure makes explicit global knowledge available to each processor, and also allows useful assumptions to be made based on implicit global knowledge. While a vertically-oriented model such as MPI is forced to make conservative assumptions about communication, *BSPlib*'s transport protocols can make much more aggressive assumptions which pay off in increased performance.

We have shown how both implicit and explicit global information can be used to: design an improved error recovery technique (hole-filling); reduce acknowledgement packet traffic by using expected traffic information; and reduce acknowledgement packet traffic further by using BSP's superstep structure. All of these features result in a transport protocol whose performance is comparable with low-latency protocols that do not address reliability. Furthermore, this low-level performance translates to improved performance for application programs.

Acknowledgements: The work of Jonathan Hill was supported in part by the EPSRC Portable Software Tools for Parallel Architectures Initiative, Research Grant GR/K40765 "A BSP Programming Environment", Oct 1995-Sept 1998. 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] 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.

[2] 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.

[3] P. Buonadonna, A. Geweke, and D. E. Culler. Implementation and analysis of the Virtual Interface Architecture. In *SuperComputing'98*, 1998.

[4] 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.

[5] S. R. Donaldson, J. M. D. Hill, and D. Skillicorn. Predictable communication on unpredictable networks: Implementing BSP over TCP/IP and UDP/IP. *Concurrency Practice and Experience*, to appear.

[6] S. R. Donaldson, J. M. D. Hill, and D. B. Skillicorn. Performance results for a reliable low-latency cluster communication protocol. In *PCNOW Workshop at IPPS'99*, 1999.

[7] J. M. D. Hill, W. 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.

[8] 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.

[9] 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.

[10] D. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.

[11] 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.

[12] M. Verma and T. Chiueh. Pupa: A low-latency communication system for Fast Ethernet. In *Workshop on Personal Computer Based Networks of Workstations, 12th IPPS/SPDP*, April 1998.

[13] 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.

[14] 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.