cBSP: Zero-Cost Synchronization in a Modified BSP Model

Richard D Alpert*
James F Philbin

NEC Research Institute
4 Independence Way
Princeton NJ 08540
{alpert|philbin}@research.nj.nec.com

Abstract

The Bulk Synchronous Parallel (BSP) model of computing proposed by Valiant [Val90] *axiomatically assumes inexpensive global synchronization. Typically, however, global synchronization is expensive, detracting from the appeal of BSP on machines without special synchronization hardware.*

Decreasing the cost of global synchronization is critical to improving the performance of any BSP implementation. We address this problem by exploiting local knowledge of the state of remote processes that can be inferred from the number of messages received from the remote process. We found that frequently, synchronization may be effected with no network traffic overhead.

This paper describes the design, implementation, and performance of a counting Bulk Synchronous Parallel (cBSP) runtime library incorporating these ideas.

1 Introduction

In parallel computing, the gap separating theory and practice is wider than in sequential computing. Most theoretical models of parallel computing were designed to study algorithmic complexity [FW78, Gib89, KLadH92] and, like the physics students' frictionless plane, do not reflect the state of real-world technology. While useful for theoretical exploration, these models of parallel computing do not incorporate the limitations and features of existing parallel machines.

Real parallel programming always has been based on specific types of application programming interfaces (APIs). These APIs have been designed around experience and need, integrating techniques of concurrent programming (e.g. Mutual Exclusion, Monitors, Threads) and interprocessor communications (e.g. PVM, RPC, NX, MPI) [BDG+91, BN84, Pie94, SOHL+95]. Because these APIs are not models of computation, it is difficult to use them to study or to predict the behavior of parallel algorithms running on parallel machines. Recent efforts such as the LogP model [CKP+92] can be used to study the behavior of message passing programs quite realistically, but they are not designed for programming parallel algorithms.

A model of parallel computing analogous to the von Neumann model of sequential computing could wed the tools of theoretical analysis to real parallel programming. The BSP model was introduced by Les Valiant [Val90] as just such a bridging model, linking architecture and software, theory and practice.

BSP offers both a a common abstraction toward which computer architects and compiler writers can design, and a concise model of parallel program execution enabling accurate performance prediction for proactive application design [HCB96, Kne94, RPL95].

The BSP model is described in more detail in section 2. In summary, however, BSP divides a computation into *supersteps*, delineated by global synchronization. Communication initiated during one superstep is not visible at a receiver until the start of the next superstep.

*This work was completed in part while this author was at Princeton University.

Researchers [BM93, Har, Oxf] are striving to implement run-time libraries and exploit the strengths of the the BSP model. Through these efforts, programmers can develop portable and scalable parallel programs, where the same code executes with predictable performance on shared or distributed memory systems, on clustered workstations or PCs. But these efforts still are not competitive with existing message-passing libraries such as NX, PVM, and MPI due to the difficulties inherent in making the BSP model efficient.

This paper proposes a counting Bulk Synchronous Parallel (cBSP) scheme that addresses the cost of global synchronization, allowing more efficient implementation on today's parallel architectures.

Our research shows that frequently, an application can determine locally the number of messages due to arrive during a given superstep. We have implemented and tested counting synchronization and compared its performance to that of global synchronization. Exploiting this knowledge can enhance application performance. cBSP continues to embrace the simplicity of the original BSP model, so it still can be used to study theoretical properties of parallel algorithms.

We describe the BSP model in section 2. Section 3 describes our research platform. In section 4, we present our rationale for modifying BSP, describe our enhancements, and analyze our design decisions. Section 5, details our implementation and the tradeoffs we made. In section 6, we describe the performance of our library, and in section 7, summarize our findings and discuss some related open research questions.

2 BSP

2.1 The BSP Model

In 1990, Valiant introduced the BSP model [Val90] to bridge the gap between the theory and the practice of parallel computing. The BSP model provides an abstraction toward which both compiler writers and computer architects can aim. One goal of BSP is its use by parallel algorithm designers as a portable programming model and as a tool to analyze and predict the performance of parallel algorithms on real hardware. Another is to allow computer architects to design efficient parallel machines without undue concern for the nature of the software that they will run, and to allow application writers to design programs that can be executed efficiently with minimal concern for the underlying hardware.

A BSP computer is one having

1. some number of *components*, each performing processing and/or memory functions,
2. a *router* that delivers messages point-to-point between components, and
3. an efficient means of *synchronizing* all or a subset of the components.

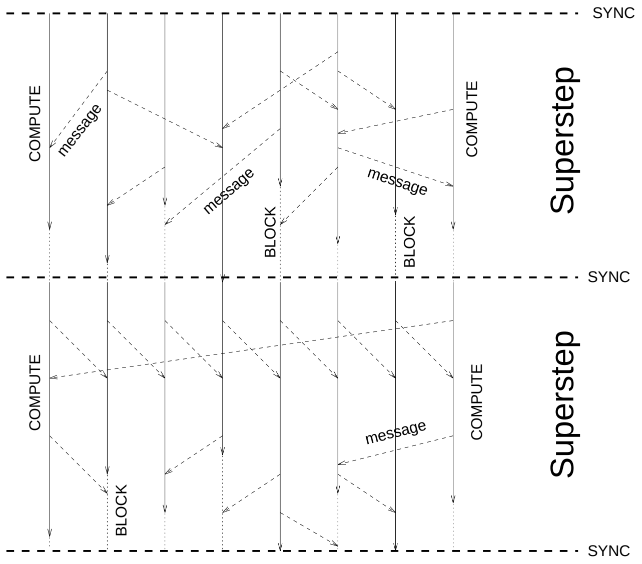


Figure 1: *Eight nodes executing two supersteps.*

The execution of a BSP program is divided into distinct phases called "supersteps". During any superstep, computation, memory references, I/O, and communication can occur. Supersteps are delineated by a synchronization operation. Data transmitted during a superstep become available to the receiver only during the following superstep. Figure 1 shows eight nodes executing two supersteps of a BSP program. Computation is shown as solid lines, communication as dashed lines, blocking as dotted lines, and synchronization as horizontal dashed lines. The blocking is a symptom of load imbalance.

The meaning of *bulk synchronous* becomes apparent. Communication is synchronous "in bulk," that is, at the synchronization points that define supersteps.

2.2 Quantifying Performance

Four parameters describe the performance of a BSP computer:

1. p, the number of processors

2. L, the synchronization cost
3. g, the communication cost, and
4. s, the processor speed.

The synchronization cost, L, is the time required to reach barrier synchronization across all p nodes, measured in processor operations.

The communication cost, g, is the least upper bound of the cost (in elapsed processor operations) of communicating a given amount of data to each processor (the least upper bound is used to account for possible network congestion). The value of g is equal to the quotient of the total number of operations performed by all processors in any given unit of time and the total amount of data that can be delivered to all processors in that same amount of time, or "aggregate processor operations per word transferred".

L and g are expressed in terms of processor operations to insure valid comparisons.

A clear advantage of this model is its simplicity. Any parallel computation becomes phases of computing, memory access, and communication, defined by synchronization points in an uncomplicated form. Because of this simplicity, the BSP model enjoys the potential to be implemented on many different parallel architectures. The model makes no assumptions about the topology of the interconnect of a parallel architecture.

BSP is useful not only to algorithm designers, but also to programmers and architects. Adhering to the BSP model permits advances in hardware and software to occur fairly independently of each other [GRT94].

Valiant asserts that the BSP model achieves both portability and efficiency for a large class of problems. The cost of portability is that efficient operation typically requires a larger input size for BSP code than for machine-specific code.

A formal analysis of some BSP algorithms can be found in [GV92]. A rather nice example of how a programmer can use the parameters p, L, g, and s, to optimize code is "parameterizing for performance," given in [MR94]: Depending on the values of the performance parameters, it may be less costly to broadcast data to processors linearly than to broadcast logarithmically.

With values of p, L, g, and s for a specific machine available at compile time, preprocessor directives could cause compilation of that code using the most efficient method to achieve some goal.

3 The Test Platform

Our implementation of cBSP exploits *virtual memory-mapped communication*.

Virtual memory-mapped communication (VMMC) occurs when a region of one process's virtual address space is mapped onto a region of another process's virtual address space. Writes then propagate with little or no sending processor involvement, and no receiver involvement whatsoever. The VMMC mechanism provides good support for protected, user-level message passing, user-level buffer management, and zero-copy protocols [DIFL96, FAB$^+$96].

Our test platform consists of 16 commodity dual 200 MHz P6, PCI-bus-based PCs, running a slightly modified Linux v2.0.24, connected with M2M-Dual-SW8 Myrinet interconnects [BCF$^+$95, Myr96].

Under VMMC, receivers grant permission for remote writes into their virtual address spaces by `export`ing regions of their virtual memory. Senders then `import` memory regions to which they wish to write. Protection is verified at the time of import.

Once the mapping has been created, a send is initiated by a few user-level instructions, instructions which cause the Myrinet DMA hardware to initiate data transfer. Buffer management is performed by senders. Data are written directly into the address space of the receiver; there is no explicit receive instruction.

Details of the VMMC implementation can be found in [DBLP97].

An important feature of virtual memory-mapped communication is that it not only supports, but requires, user-level buffer management. Libraries and user programs customize their own buffer management to implement zero-copy protocols, to achieve very low-latency message passing, and to exploit the raw bandwidth available in hardware. One challenge in implementing cBSP at user level is to capitalize fully on the advantages of the underlying communication mechanism while preserving the semantics of the BSP model.

4 cBSP Design

4.1 Identifying Cost

Because of the importance of global synchronization in the BSP model and its typical cost ($O(\log P)$ time, $O(P)$ messages), the challenge in creating an efficient implementation is to design a scheme for which, in the common case, the cost of global synchronization can be minimized or avoided entirely.

Synchronization generally is viewed as a temporal constraint. At a given point in the execution of an application, processes "stop" to wait for siblings to "catch up" with them. The invariant condition of nearly all global synchronization operations is that no process exits the synchronization routine until all processes have entered that routine.

In the BSP model of parallel computation, however, synchronization satisfies a logical, yet less strict need: that of insuring correct computation. Synchronization guarantees that certain data have been sent and that certain data have arrived.

To decrease or eliminate the cost of synchronizatin in BSP is to satisfy these communication constraints using the least expensive means possible.

Examining applications, we found that there are instances in which even a sender can not determine the number of messages it is sending to a given receiver during a given superstep until that superstep has ended. Frequently, however, receiving processes can determine the number of messages that should arrive during a given superstep, and thus derive some information about the state of the computation at the sending node without resorting to additional communication.

This derived knowledge of a sender's computational state can be used by a receiver to lower the cost of determining superstep boundaries.

4.2 Synchronization

In four applications written for BSP (Gaussian Elimination, Ocean, Barnes-Hut N-Body, and LU Decomposition), processes could determine the number of messages to be received a large percentage of the time.

In gauss and ocean, receivers could use counting synchronization in 100% of the supersteps. LU decomposition showed slightly lower percentages. Values for each program in the suite are shown in Figure 3.

In ocean and LU decomposition, however, a few simple calculations were necessary, as the number of messages is a function both of a node's position in the grid and the number of columns and rows in the subarray being processed by the node at the time.

After the requisite number of messages has been received, a process can continue into the following superstep without imperiling program correctness. To prevent data corruption, our library cycles through disjoint data structures in consecutive supersteps. With a few such data structures, communication patterns themselves enforce an adequate (safe) level of synchrony.

Exploiting these characteristics enables local end-of-superstep synchronization without introducing additional network traffic. The key is exploiting information about the state of a computation at a remote process conveyed by the *number* of messages sent by that process.

Avoiding network synchronization traffic is not totally free of cost, however. A receiving node must be able to count the number of messages that actually arrive during a given superstep. Counting can be accomplished by buffering messages at the receiver and counting them as they are copied into their final destinations. Alternatively, at synch time, a sender can inform a receiver explicitly of the number of messages it sent.

4.3 The cBSP API

The goal of this research is not the creation of an ideal API for BSP, but to test specific features, principally the feasibility of effecting synchronization locally. There is a significant effort underway to create a world-wide standard BSP API [GHL$^+$]. We do not suggest that our API can or should compete with this effort.

We chose to keep our API as simple as possible. It consists of these four functions:

- `cBSP_initialize()`
  creates, allocates, and initializes buffers, assigns logical node numbers

- `cBSP_send(node, destAddr, srcAddr, nbytes, handler)`
  causes nbytes of the contents of local virtual address `srcAddr` to be written to remote virtual address `destAddr` on logical node `node`. The write is guarantee to occur by the beginning of the next superstep.

  If non-NULL, `handler` will be executed on logical node `node` synchronously, at user-level, at the time of synchronization on `node`, after the accompanying data have been written to their destination.

  The handler takes three parameters: the source node, the destination address, and the size of the message in bytes. Its prototype is

  `void handler (int srcNode, void * destAddr, int nBytes)`.

  A NULL (zero) destination address in a cBSP_send call has special significance. An application programmer may prefer to operate on data directly from an incoming message buffer without first copying into the applications address space. Scatter/gather is one such case. The programmer supplies a handler and specifies a NULL destination address. The handler is passed the address of the data in the receive buffer, rather than a destination address in the user's virtual address space.

- `cBSP_nsync(count)`
  prevents a process from proceeding until `count` messages have been received and written to their destination addresses. If any handlers have been specified, they will have executed to completion prior to the exit from `cBSP_nsync`. There are no guarantees with respect to execution order of multiple handlers, nor of multiple sends to the same destination address.

- `cBSP_gsync()`
  is a global synchronization. Any pending messages are guaranteed to have been written, and any handlers specified are guaranteed to be executed prior to exit from `cBSP_gsync`. No process exits `cBSP_gsync` until every process has entered.

From these four primitives, broadcast and reduction functions can be built in a straightforward manner.

There is no explicit receive function. Data are guaranteed to have been written by the beginning of the next superstep. In our library, all writes actually occur during synchronization.

5 cBSP Implementation

5.1 Initialization and Use of Buffers

At initialization time, each process allocates a region of memory for messages from each other process.

Because message buffers are completely cleared at the end of each superstep, there is no need for a buffer to have a strict, regular format. This contrasts with the VMMC implementation of NX/2 [ADFL96, Pie94]. Because NX messages can be consumed out-of-order, and because there is no time at which it is known that a message buffer will be completely empty, NX messages are sent to clearly defined "slots" in the message buffer.

In cBSP, messages of different sizes fill the buffer space as needed with almost no waste (a very small number of bytes are sacrificed to satisfy alignment constraints). At the beginning of each superstep, senders are free to (and are expected) to refill message buffers from the beginning.

Messages are written into buffers from higher to lower addresses. This scheme was chosen because the VMMC library delivers data from lower-to-higher address; receivers can examine the last data delivered to sense message arrival.

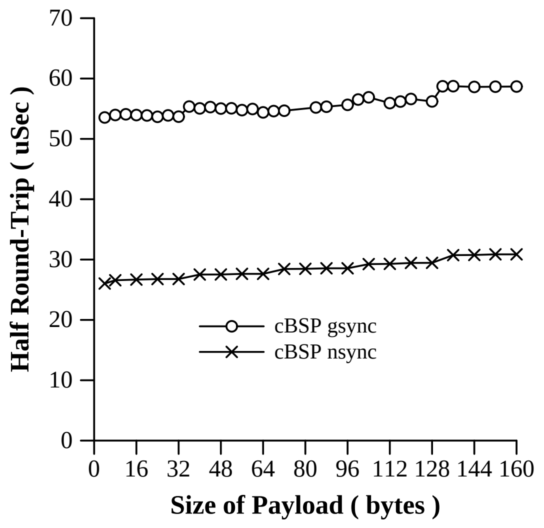
The layout of a message in the buffer is straightforward. For each message, there is an arrival flag, a size field, the data, destination address, handler, and the superstep number. The superstep number modulo some integer (the number of sets of such buffers) provides an index into the receive buffer data structures.

5.2 Flow Control

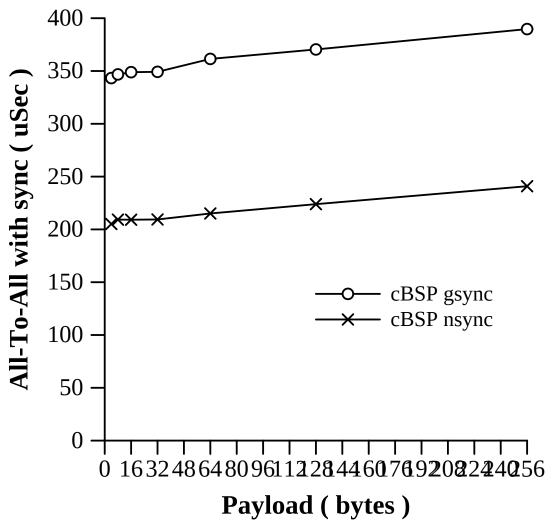
When messages are buffered at the receiver, some provision must be made for buffer overflow. Because communication is sender managed, it is the sender, not the receiver, that detects when an overflow condition is imminent.

If a message would cause a receive buffer to overflow, the sender writes a special overflow flag in the `size` field of the final remaining message region. Subsequent messages for the current superstep are then buffered at the sender in a backup buffer.

When the receiver is synchronizing, it sends a special message to the sender after it has consumed all of the messages in its receive buffer and senses the overflow flag. The sender then sends the contents of the backup buffer to the



(a) *"Ping-Pong" Latency*



(b) *"All-to-All" Latency*

Figure 2: *Small Message Latencies*

receive buffer, and the receiver continues. The sender cannot complete synchronization until it has satisfied all pending requests for backup buffers.

5.3 Large Messages

If a message is larger than an entire message buffer, a special flag is written in the `size` field and the message is buffered on the sender side. As with small message flow control, the receiver, during synchronization, sends a special message to the sender, clearing the sender to write the data directly to its final destination.

6 Performance

The cBSP libraries were first used to obtain microbenchmarks for small message latency. These results were intuitive, with global synchronization requiring approximately twice as much time as counting synchronization. To draw a more practical picture of the relative performance of the two schemes, applications having different communication styles were run using the different protocols.

6.1 Microbenchmarks

Small message latency performance is shown in the graph of Figure 2, which shows the one-way latency observed in a "ping-pong" test. Two processes send payloads to each other, with data movement alternating direction in consecutive supersteps. For comparison, eight-node all-to-all latency also is show in Figure 2.

Using `cBSP_gsync`, naive global synchronization is effected by an $O(\log n)$-time, $O(n)$-messages technique, with explicit synchronization messages passing between nodes.

The `cBSP_nsync` scheme sends no messages other than those containing the actual data. When the prescribed number of messages has arrived (in this case, one), the process proceeds. As one might intuit, latency is about half that of naive global synchronization.

6.2 Testing with Applications

We tested counting vs. global synchronization on `gauss`, a Gaussian Elimination solver, `nbody`, Barnes-Hut N-Body, `lu`, LU Decomposition using grid distribution, and `ocean`, simulating eddy and boundary currents and large-scale ocean movements.

The communication behaviors displayed by these applications are varied, and are typified by `gauss` and `nbody`.

Communication in `gauss` is regular and predictable, with message counts and message sizes known in advance, making `gauss` a good candidate for counting synchronization. Knowing only the dimensions of the matrix, it is trivial to determine the time, size, source and destination of each message prior to program execution.

`LU` decomposition is almost as predictable, but there are a few points at which some short calculations (mostly boolean) must be performed to determine the number of incoming messages.

Application	Supersteps	Percent Known	Speedup
gauss	1025	100.00%	1.116
LU	2564	79.95%	1.052
ocean	2634	100.00%	1.139
n-body	180	26.11%	1.180

Figure 3: Application Characteristics

In Barnes-Hut `N-Body`, communication is irregular and dependent upon the nature of the data. Three-dimensional space is represented by an octtree, and is hierarchically divided according to the density of bodies within each region. As the algorithm progresses and bodies are displaced, they may be transferred from one process to another. Each movement of a body from one process to another occurs in a separate message.

As a result, `nbody` is less able to exploit counting synchronization, though message counts are known at some stages of the algorithm. `N-body` was the only one of our applications for which we could not determine the number of incoming messages in a significant fraction of the supersteps. Simulating 16000 bodies through 20 iterations, the number of incoming messages can be determined in only 47 of 180 supersteps (26.11%). In many of these supersteps, not only can senders not determine the number of messages they are sending to a given node until synchronization time, but the destination addresses of these messages can be determined only by receivers. This occurs during the exchange of bodies between nodes, because senders lack sufficient information about the state of the receivers' data structures.

Applications were run on the test platform (Sec 3) using both global and counting synchronization. The results are shown in Figure 4. The vertical axis of Figure 4 represents relative computation time, with the longest time as 100 (shorter is better).

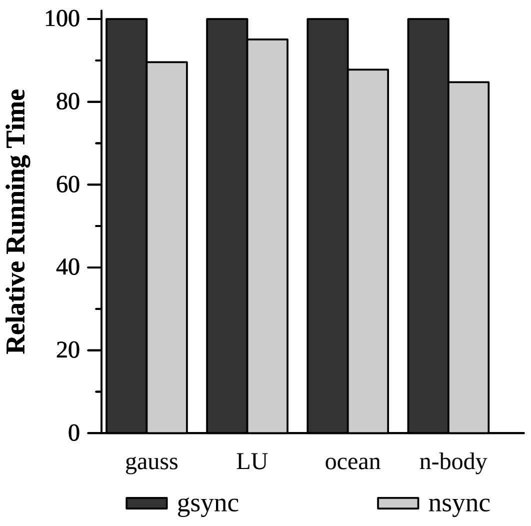


Figure 4: *Relative Running Time*

In Table 3, which summarizes characteristics and observed behavior of the applications, shows, for each application, the number of supersteps, the percentage of supersteps for which the number of incoming messages can be determined by the receiver, and the observed speedup using counting synchronization over global synchronization.

6.2.1 Gaussian Elimination

Computation and communication both are regular and predictable in `gauss`, making it an ideal candidate for counting synchronization. The number of messages expected at each stage of the computation is fixed and can be "hardwired" into the code. On 8 nodes, solving a 1024-equation system, `gauss` used 1025 supersteps in about 11 seconds. Counting synchronization showed a speedup of 1.116 over global synchronization.

6.2.2 N-Body

Barnes Hut N-Body [BH86] is a hierarchical $O(N \log N)$ force calculation algorithm. Bodies, each having different mass, acceleration, and initial velocity, are distributed in space. Space is then recursively partitioned, forming an octree. Bodies are at the leaves; regions of space are represented by interior nodes. Different regions in space then are assigned to different processors.

Rather than individually calculating the result of the bodies' mutually attracting forces (an $O(N^2)$ algorithm), bodies sufficiently distant are grouped and assumed to have a mass and centroid equal to the sum of the masses and sum of the centroids of the individuals, producing an $O(N \log N)$ algorithm.

The shape of the octree is a function of the distribution of the bodies in space, and the distribution changes as the algorithm progresses, so it is not possible to predict communication needs and behaviors. This property of the algorithm limits potential speedup of counting synchronization.

Simulating the motion of 16000 bodies with randomly assigned initial velocities and accelerations through twenty time steps used 180 supersteps in about 11 seconds on 8 processors. Using counting synchronization wherever possible resulted in a speedup of about 1.180 over global synchronization.

6.2.3 LU

Like Gaussian Elimination, LU Decomposition is an algorithm for solving systems of linear equations written as $Ax = b$, where A is an $n \times n$ coefficient matrix and x and b are $n \times 1$ column vectors. Unlike Gaussian Elimination, LU first factors A into a lower triangular and upper triangular matrix.

An upper [lower] triangular matrix is one in which all elements below [above and including] the diagonal are zero. Once A has been factored into L and U, solution of the system is straightforward.

`LU` is able to exploit counting synchronization, but not without incurring the small cost of a few simple calculations to determine the number of incoming messages.

On 8 nodes performing LU decomposition on a 512×512 array used 2564 supersteps and completed in approximately 11 seconds. Using counting synchronization yielded a speedup of 1.052over global synchronization.

6.2.4 Ocean

`Ocean` models the role of eddy and boundary currents in influencing large-scale ocean movements. This implementation, modified from the SPLASH suite [SWG], uses dynamically allocated four-dimensional arrays for grid data storage.

The behavior of `ocean` meets intuitive expectations: counting synchronization is faster than global synchronization, though with results less dramatic (7% speedup) than some of the more regular applications.

7 Conclusions

Some applications can benefit from counting synchronization. Factors favoring use of counting synchronization include regularity of communication and larger numbers of processors

For the programs in our suite, speedups using counting synchronization ranged from 1.052 to 1.180.

Porting existing applications to cBSP was fairly straightforward, with determining the number of incoming messages the only sometimes non-trivial task.

Because a sender usually is unaware of the scope of a receiver's data structure, sometimes destination addresses can not be determined by a sender. cBSP's handler feature was easily powerful enough to cope with those situations where receivers determined destinations.

8 Acknowledgements

Cezary Dubnicki designed and implemented the Paragon VMMC library.

The original idea for counting synchronization grew out of discussions with Kai Li.

The LU decomposition code was provided by Bob Bisseling, via his web page [Bis].

The flow control buffer management scheme was developed during a conversation with Stefanos Damianakis.

Richard Alpert has enjoyed the partial support of an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland.

References

[ADFL96] Richard D Alpert, Cezay Dubnicki, Edward Felten, and Kai Li. The Design and Implemenation of NX Message Passing Using SHRIMP Virtual Memory Mapped Communication. In *Proceedings of 25th International Conference on Parallel Processing*, pages I–111 – I–119, August 1996.

[BCF+95] N J Boden, D Cohen, R E Felderman, A E Kulawik, C L Seitz, J N Seizovic, and W Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[BDG+91] A Beguelin, J Dongarra, A Geist, R Manchek, and V Sunderam. A Users' Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN, July 1991.

[BH86] J. E. Barnes and P. Hut. A hierarchical $O(NlogN)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[Bis] Bisseling. BSP Dense LU Decomposition. http://www.math.ruu.nl/people/bisseling.html.

[BM93] R H Bisseling and W F McColl. Scientific Computing on Bulk Synchronous Parallel Architectures. Technical Report 836, Department of Mathematics, University of Utrecht, Dec 93.

[BN84] A D Birrell and B J Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[CKP+92] D Culler, R Karp, D Patterson, A Sahay, K Schauser, E Santos, R Subramonian, and T von Eicken. LogP: towards a realistic model of parallel computation. Computer science division report, University of California, Berkeley, 1992.

[DBLP97] Cezary Dubnicki, Angelos Bilas, Kai Li, and Jim Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *11th International Parallel Processing Symposium*. IEEE Computer Society Technical Committee on Parallel Processing, 1997. To Appear.

[DIFL96] Cezary Dubnicki, Liviu Iftode, Edward Felten, and Kai Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[FAB+96] E Felten, R Alpert, A Bilas, M Blumrich, D W Clark, S Damianakis, C Dubnicki, L Iftode, and K Li. Early Experience with Message-Passing on the Shrimp Multicomputer. In *International Symposium on Computer Architecture XXIII*, 1996.

[FW78] Steven Fortune and James Wyllie. Parallelism in Random Access Machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118. Association for Computing Machinery, 1978.

[GHL+] Mark W Goudreau, Jonathan M D Hill, Kevin Lang, Bill McColl, Satish B Rao, Dan C Stefanescu, Torsten Suel, and Thanasis Tsantilas. A Proposal for the BSP Worldwide Standard Library. http://www.bsp-worldwide.org/standard/stand2.htm.

[Gib89] P B Gibbons. A More Practical PRAM Model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168. Association for Computing Machinery, 1989.

[GRT94] Mark W Goudreau, Satish B Rao, and Thanasis Tsantilas. A Study of the BSP Model: Algorithms and Implementation. Technical Report UCFCS:CS-TR-94-04, Department of Computer Science, University of Central Florida, April 1994.

[GV92] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. Technical Report TR-10-92, Harvard University, Computer Science Department, 1992.

[Har] Harvard University. The Harvard BSP Project. http://www.das.harvard.edu/cs/research/bsp.html.

[HCB96] Jonathan M D Hill, Paul I Crumpton, and David A Burgess. The theory, practice, and a tool for BSP performance prediction applied to a CFD application. Technical Report TR-4-96, Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, England. OX1 3QD, February 1996.

[KLadH92] R M Karp, M Luby, and M Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. In *Proceedings of the Twenty-Fourth Annual Symposium of the Theory of Computing*, pages 318–326. Association for Computing Machinery, May 1992.

[Kne94] Simon Knee. Program development and performance prediction on BSP machines using opal. Technical Report PRG-TR-18-1994, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 1994.

[MR94] Richard Miller and Joy Reed. *The Oxford BSP Library Users' Guide*. Oxford Parallel, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, 1.0 edition, May 1994.

[Myr96] The Myrinet on-line documentation. http://www.myri.com:80/scs/documentation, 1996.

[Oxf] Oxford University. The Oxford BSP Group. http://www.comlab.ox.ac.uk/oucl/groups/bsp/research.html.

[Pie94] Paul Pierce. The NX Message Passing Interface, Parallel Computing Vol. 20 no. 4, April. *Parallel Computing*, 20(4), April 1994.

[RPL95] Joy Reed, Kevin Parrott, and Tim Lanfear. Portability, predictability and performance for parallel computing: BSP in practice. November 1995.

[SOHL+95] Marc Snir, Steve W Otto, Steven Huss-Lederman, David W Walker, and Jack Dongarra. *MPI The Complete Reference*. MIT Press, Cambridge, Massachusetts, 1995. ISBN 0-262-69184-1.

[SWG] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.