CCIT Report #588 May 2006

BSP-IB: the Foundation of a Well-Balanced, Programmer-Friendly Commodity Supercomputer

Yitzhak Birk, Yoel Davidson, Dor Ganor, Liran Liss, and Assaf Schuster
Technion – Israel Institute of Technology
{birk@ee,sdayoel@t2,sdorg@t2,liranl@tx,assaf@cs}.technion.ac.il

June 24, 2006

Abstract

The Bulk Synchronous Parallel (BSP) programming model is attractive for parallel programs that proceed in "phases": it is simpler to program than more general message-passing models such as MPI, and facilitates performance forecasting. While restricting overlap of communication and computation, it lends itself to efficient communication through aggregation and scheduling, thereby preventing unnecessary endpoint contention. InfiniBand is a standard high-speed, low-latency interconnect for clusters. This paper explores the possibility of efficient BSP implementation on computer clusters. We present an architecture and prototype implementation of BSP in an InfiniBand-connected PC-based cluster, utilizing the unique capabilities of InfiniBand for efficient implementation of key BSP functions such as barrier synchronization. This enables demanding applications to achieve good speedups. Our commodity BSP-IB cluster forms a well-balanced parallel machine, outperforming custom multiprocessor BSP machines (e.g., IBM SP and Origin 2000) in every respect.

1 Introduction

In parallel programs, it is desirable to permit computations to proceed as flexibly and aggressively as possible while maintaining correctness. Consequently, message-passing models like MPI [1] that permit a processor to send results to another processor as soon as they become available appear very attractive. Unfortunately, however, this flexibility can only be exploited if the programmer takes upon himself much of the responsibility for avoiding incoherence. (The communication libraries offer mechanisms that assist in so doing, but the programmer is charged with deciding when reduced synchronization may be allowed.) Also, many parallel programs are structured in phases, be it because of the very nature of the algorithms being programmed or as a means for the programmer to manage complexity. In each phase, a processor computes intermediate results over a subset of the data, and subsequently exchanges results with some subset of the processors. In phase-based programs, the

potential benefit of permitting processors to "run ahead of the pack" is limited. Specifically, the benefit from overlapping computing and communication is bounded from above by a factor of two.

1.1 Bulk-Synchronous Parallel (BSP) Programming Model

The Bulk Synchronous Parallel (BSP) model, proposed by Valiant in 1989 [2], is a general parallel programming paradigm for phase-based programs. It can apply to any group of processor-memory pairs interconnected by a communication network. In BSP, parallel programs are executed in synchronous phases, called supersteps. During a superstep, every node performs local computation and posts requests for data transfer. However, the effects of data transfers are generally made visible to the processors only after a well defined global barrier synchronization point is reached by all nodes, which ends the superstep. BSP has only 20 primitives [3], as compared with nearly 300 for MPI. Most of the communication is effected using one-sided operations, put and get, causing BSP to resemble a scalable shared memory model rather than a message-passing interface. High performance versions (with weaker semantics) of all communication primitives are also supported. BSP programs are structured, easy to write, and easy to debug.

BSP's synchronous operation also offers a convenient cost model that characterizes an architecture using only 4 parameters: the number of processors p ; the processor speed r [flops/sec]; the normalized (by processor speed) barrier latency l [flop]; and the normalized (inter-processor) communication-time per word in continuous traffic (i.e., 1/BW) g [flop]. By assuming that contention occurs at the network end-points and that sends do not contend with receives, superstep time estimates can be calculated using the so called notion of an h-relation, which stands for the maximum over all nodes of *max*{*#words-sent, #words-received*}. This assumption has been validated for numerous architectures [4]. Consequently, if an algorithm executes at most x Flops per processor (given a certain number of processors) in a y-relation superstep, then the superstep's execution time is: $(x + yg + l)/r$ seconds.

BSP enables implementations to amortize the overheads of individual data-transfer requests by aggregating messages together during computation, and performing bulk data transfers at barriers. Also, it prevents communication bottlenecks by carefully scheduling data transfers [5]. The benefits of these optimizations have been shown to often more than offset the potential speedup of 2 obtained by overlapping communication and computation. (For the more flexible approach of MPI to offer a two-fold speedup in phase-based applications, there must be perfect balance between computation and communication in every phase, and results must become available at a rate that at least equals the progress of the computation.) High performance BSP implementations [6, 7] have been made available for a wide range of multi-processor machines, such as Cray T3E, IBM SP, and SGI Origin 2000.

1.2 Cluster-Based Parallel Computers

With the advent of System Area Networks (SANs), which exhibit data rates in excess of 10Gbps and microsecond-scale latencies, commodity computing clusters offer a cost-effective alternative to expensive special purpose parallel machines. Clusters based on SANs such as Quadrics [8], Myrinet [9], and InfiniBand [10] dominate the top 500 supercomputer list, and have become the de facto standard for high performance computing. These clusters are typically programmed with proven, full-featured, message-passing industry standards such as MPI [1], which often require great expertise to carefully choose the correct primitives for good performance. Thus, while delivering superb performance, message-passing programs are often hard to develop and debug.

1.3 BSP on Cluster-Based Parallel Computers

In the past, BSP implementations for clusters suffered from relatively high synchronization costs and limited bandwidth compared to multi-processors [4]. With the advent of high-performance SANs, the questions that arise naturally are 1) whether the simple BSP programming model can be efficiently applied to SAN-based clusters, and 2) whether SANs can be used to provide a commodity BSP machine with supercomputer performance.

It appears that SAN features such as reliable connections, user-level networking and Remote Direct Memory Access (RDMA) can be employed to considerably improve BSP performance. Indeed, a recent implementation of BSP over GigaNet's cLAN interconnect, xBSP [11], has shown that BSP's "high-performance" one-sided communication primitives deliver nearly raw hardware performance. However, these "high-performance" primitives do not obey the strong superstep semantics, i.e., data is not buffered at the sender or at the receiver, and the effects of data-transfers can become visible to the processors before the end of the superstep. (Note that high-performance one-sided operations are not unique to BSP; they are also an inherent part of the MPI-2 standard.) The question whether the core BSP operations that offer the simplicity to the programmer can be implemented efficiently has thus remained open.

In this paper, we present efficient implementations of the main vehicle for data transfer in BSP, i.e., one-sided (buffered) communication operations. We have designed and implemented from the ground up a high-performance BSP prototype over Linux for InfiniBand SANs, BSP-IB, taking advantage of new SAN capabilities and BSP's special characteristics whenever possible. Our baseline implementation avoids interrupts and system calls completely, conducts all data transfers of a superstep using as little as a single I/O operation per send-receive pair, and cuts down the work per data-transfer request to a bare minimum. Further optimizations include utilizing raw hardware multicast performance via a Nack-based protocol to reduce synchronization time, making use of IB's hardware scatter-gather capabilities to achieve efficient 0-copy data transfers (on the receive side), and reducing control bandwidth in fine-grain data transfers.

Our BSP-IB PC-based prototype cluster features higher computing rate and bandwidth, and lower synchronization latency than multi-processor machines such as SGI Origin 2000,

IBM SP, and Cray T3E. More importantly, BSP-IB is also a balanced machine: after factoring out the processor speed by normalization, BSP-IB's g and l architectural parameters are also better than some of these machines. Finally, BSP-IB provides excellent speedups for important applications such as FFT and LU decompositions.

The remainder of this paper is organized as follows. Section 2 provides a short overview of BSP and InfiniBand. In section 3, we describe in detail our baseline BSP design and implementation along with several optimizations. Performance results are presented in Section 4, and Section 5 offers concluding remarks.

2 Preliminaries

This section presents brief overviews of InfiniBand and BSP, providing the information necessary for understanding the architecture and prototype described in the next section.

2.1 InfiniBand and uDAPL

InfiniBand (IB) [10] is a high-performance SAN architecture that provides data rates in excess of 10Gb/s as well as several-microsecond latencies. IB implements in hardware protocols for reliable connections, multiplexing, and flow-control. These features, along with virtual to physical memory translation capabilities, enable IB to support user-level APIs that do not require operating system intervention in the fast path. InfiniBand supports both message passing and remote direct memory access (RDMA) semantics. (In RDMA, a host can read or write remote memory without involving the remote CPU by specifying memory locations at both ends.) One example for such an API is the user-level Direct Access Programming Library (uDAPL) [12]. Hereafter, we will use uDAPL terminology when referring to IB abstractions.

IB achieves its best performance with asynchronous I/O. Once a connection between two Endpoints (hosts) is established, applications typically post Data Transfer Operations (DTOs) to the hardware, which are then processed asynchronously. DTOs either point to full (for sends and RDMA writes) or empty (for receives and RDMA reads) buffers. All such buffers must reside in Local Memory Regions (LMRs) that were explicitly registered with the hardware beforehand. Registration ensures that LMRs are pinned to physical memory and that their physical addresses are known to the hardware. Completed operations can be detected by polling a Completion Queue, which can aggregate completion events for multiple connections. Blocking DTO requests and asynchronous completion notification are also supported. Any buffers associated with outstanding DTOs are considered to be under the responsibility of the InfiniBand channel adaptor and should not be accessed by the application. The foregoing scheme thus results in 0-copy on both the send and receive sides, enabling wire-speed data-transfers with minimum overhead to the CPUs, memory systems and internal data paths.

2.2 The Bulk Synchronous Parallel (BSP) Standard

The BSP standard [3] includes 20 primitives. Data transfer requests are effected mainly by one-sided operations, namely bsp_put and bsp_get. A form of bulk message-passing is also supported, but is designed mainly for applications with irregular communication patterns. Barrier synchronization is achieved by calling bsp_synch. BSP defines an additional high-performance version for all data transfer operations, e.g., bsp_hpput and bsp_hpget. These primitives achieve efficient unbuffered communication, at the expense of violating the conventional BSP semantics of revealing communicated data only after the next barrier synchronization. In order to expose local memory to remote access and facilitate naming, BSP employs a registration scheme that binds (usually the same) variables on different nodes. Thus, one-sided communication operations can reference remote variables by their local counterparts. A variable is registered using bsp_push_reg and deregistered using bsp_pop_reg. All registration changes take effect after the next barrier synchronization in accordance to BSP semantics.

3 Our BSP-IB Design and Implementation

bsp_put is by far the most widely-used BSP primitive [4], due to its intuitive sender-initiated operation and strict compliance with BSP semantics: data sent with bsp_put is buffered at the sender to enable the user to reuse the buffer immediately, and is guaranteed to be made visible at the receiver only after the next superstep, regardless of when the data actually arrived. Therefore, we have decided to focus on the bsp_put primitive as a faithful representative of most BSP communications. Also, an efficient implementation of bsp_put can serve as a good reference for the other primitives, such as bsp_get. (In fact, the Oxford BSP toolset [6] actually implements bsp_get using bsp_put.) Specifically, we concentrate on optimizing fine-grained invocations of bsp_put, which can accommodate the most demanding communication patterns. We next detail our baseline implementation followed by several optimizations.

3.1 Baseline Design and Implementation

Our implementation builds upon the important lessons learned from previous BSP implementations [13]. Specifically, we pack multiple data transfer requests (individual bsp_put calls) of a given sender targeting the same destination node during computation, and send them en-masse during barrier synchronization. The packed data is partitioned into fixed size chunks, which are sent according to a Latin-square schedule [5] to prevent congestion at end nodes. As in Oxford's BSP toolset [6], we associate with every registered variable an identifier, which is incremented after every registration request. (Assuming that BSP registrations are conducted in the same order in every node, variable IDs are unique.) When a remote variable is referenced by a local address during a data transfer request, this address is translated to a global identifier, which is sent with the data. Subsequently, the identifier is translated to the actual remote variable address during data placement at the receiver.

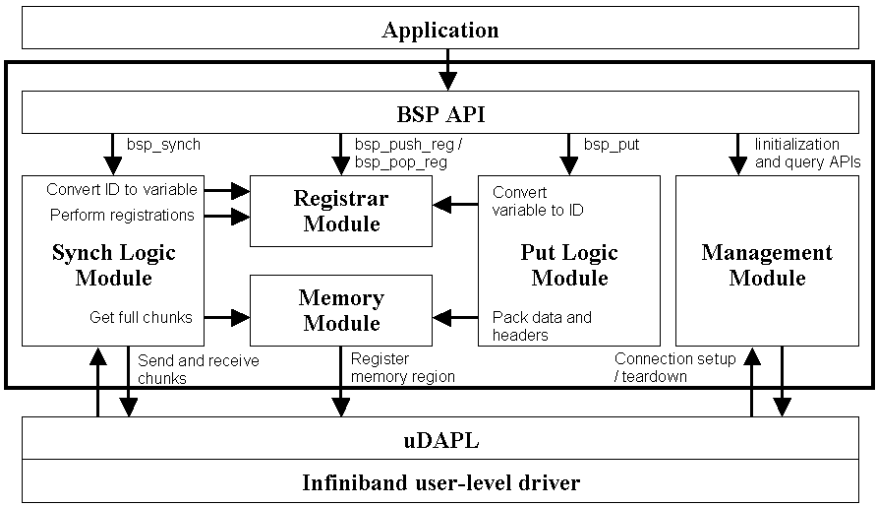

Figure 1: BSP-IB block diagram.

Our implementation takes advantage of new SAN capabilities while minimizing overheads using BSP's special characteristics whenever possible. It consists of the following logical components, depicted in Figure 1: (1) a management module, which implements communication setup (we establish a reliable connection between every pair of hosts), communication teardown, and system information (e,g, number of CPUs, CPU rank, timer, etc.); (2) a memory module, which allocates memory, registers LMRs with IB, and manages buffers; (3) a registrar module, which implements the bsp_push_reg and bsp_pop_reg primitives, and efficient variable-to-ID and ID-to-variable conversions; (4) a "put" module implementing the bsp_put logic; and (5) a "synch" module implementing the bsp_synch logic. We next outline the flow of control among these components. Subsequently, we detail the principles that guided our implementation.

When a BSP application is started, the management module sets up the proper network connections, and the memory module allocates an initial memory region and registers it with IB as an LMR. This region is logically split into fixed size chunks that form the basic allocation unit. During a superstep, bsp_push_reg and bsp_pop_reg invocations are buffered by the registrar module. These registrations are applied when the next barrier is reached, by assigning global identifiers to variables and populating conversion hash-tables. Thus, newly registered variables are made available for data transfers in the subsequent superstep.

Calls to bsp_put are handled as follows. For every destination node, the memory module holds a list of full chunks intended for that destination (the last chunk may be partly full). Given a variable and a destination node to operate on, the "put" module initially looks up the variable's global ID using the registrar and prepares a short control header, which is appended to the last chunk corresponding to that destination node. If the chunk is full, a new chunk is allocated from the memory module. Finally, the application data itself is copied, allocating additional chunks as necessary. In case a superstep runs out of chunks, another region is allocated and registered, and its chunks are added to the pool. Memory

regions are unregistered and freed only when the application exits.

Upon reaching bsp_synch, every node exchanges short information messages with every other node, informing it about the number of chunks that it has to send to that node. After a node has received an information message from every other node, it commences a tight loop, which attempts to alternately send a chunk to a remote node (according to a Latin-square schedule [5]) and poll for incoming data. If the send queue of a certain endpoint is full, a send operation will temporarily fail due to insufficient resources. However, once the peer node reposts a (IB) receive buffer, hardware flow control kicks in and the next pending send request is processed. For every endpoint, the "synch" module maintains a fixed number of receive chunks that are allocated upon startup. Full received chunks are handled immediately: the data is copied into place according to the global identifiers found in the corresponding headers, and the chunk is reposted to the receive queue. The loop exits once all outgoing chunks have been sent and all expected incoming chunks have been processed. Finally, any pending registration changes are performed and the barrier exits.

It should be pointed out that requiring every node to send a control message to every other node is wasteful, though these messages are very short. This is particularly true if each node only plans to send data to very few other nodes. This will be addressed by the optimizations. The implementation adheres to the following design principles:

- **Efficient, synchronous, user-level implementation.** Using the uDAPL user-level interface, all fast-path communication, e.g., sends and receives, is initiated from user level. Completed I/O operations are detected solely by user-level polling. This is possible because the actual communication is executed synchronously at barriers, during which the processors would otherwise be idle. Moreover, all communication tasks are handled by the application processes themselves, avoiding context switches. Finally, completion events of all connections are aggregated by a single hardware completion queue, which reduces response times.

- **Minimize work per data-transfer request.** In addition to transferring the data of multiple bsp_put operations in a single message (a chunk), several measures are taken to reduce bsp_put specific costs to a bare minimum. First, the data is packed directly into the communication buffers used by the hardware at the sender, and unpacked from them directly into their final destination at the receiver. This results in a single data copy at the sender, which is mandatory according to BSP semantics, and a single copy at the receiver, which is also mandatory if the destination buffers are not registered with the hardware. (Achieving 0-copy at the receive side is discussed in the next section).

  Second, the control header that is generated by each bsp_put invocation is packed with the data to enhance caching. This information is accessed again only at the receiver side when the data is copied to its final memory location while executing bsp_synch. Thus, control information is never copied, and there are no per-bsp_put overheads during the data transfer itself.

Finally, translating a variable address to a global identifier (during bsp_put at the sender) and vice versa (during bsp_synch at the receiver) is done using an efficient O(1) hash lookup operation.

- **Amortize memory allocation and registration costs.** Since memory allocations and IB registrations are expensive operations, registered memory areas are allocated in large quantities of several MB. During bsp_puts, fixed-sized communication chunks are then allocated from these regions as needed in a simple, sequential manner with minimal overhead. Note that within a node, these chunks constitute a shared pool that serves all connections, thus achieving a flexible allocation scheme. When bsp_synch is reached, all chunks are reclaimed once their data has been transmitted.

- **Maximize throughput.** The CPU and the IB hardware do not necessarily contend on the same bandwidth to memory because their memory bandwidth can be limited by bottlenecks in different components (e.g., busses, interconnects). Therefore, during data transfers, both the IB hardware and the CPU participate concurrently in a two stage pipeline: IB transfers data between nodes into pre-allocated and registered chunks, and the CPU scatters the data within a node to its final destination. Note that BSP's separation between computation and communication ensures that all CPU resources are at our disposal during data transfer.

  To maximize the pipeline throughput, several hardware characteristics and features are taken into account: (1) every chunk is sent using a single I/O request (we fix the chunk size at 16KB, which delivers optimal bandwidth on our hardware [14]); (2) for every connection, the receiver holds a sufficient number of chunks so that the IB link can operate in full capacity as long as the CPU can keep up.

3.2 Optimizations

In addition to the baseline implementation, we experimented with several optimizations: header compression, hardware multicasts, and 0-copy receives, which are detailed below.

- **Header compression.** In the baseline implementation, each bsp_put operation packs with the data a control header of 12 bytes. For fine-grain data access (e.g., single words), these headers may pose a significant overhead in terms of both memory and bandwidth. To remedy this problem, we also implemented a compact header format that uses only 4 bytes by restricting the number of bits in the header fields (e.g., the data transfer size). Thus, every fine-grained bsp_put that obeys these restrictions can be packed efficiently with minimal overhead.

- **Hardware multicast synchronization.** To reduce barrier synchronization latency, we replaced the all-to-all exchange of information messages in the baseline implementation with a scheme based on IB's hardware multicast capabilities. Specifically, every node that reaches a barrier sends to a designated manager node an information message containing the number of chunks it has to send to every other node. The manager

node combines these information messages to a single information vector that describes the total number of incoming chunks for every node, and multicasts the vector once all information messages have been processed. After obtaining the vector, a node may proceed to the next superstep as soon as its expected chunks are received. Consequently, no node is required to perform $O(p)$ costly send operations (where p is the number of processors) as in the baseline implementation. The single node that gathers the sending plans of all nodes must still receive p messages, but the length of each such message is only proportional to the number of nodes to which its sender plans to send data. Also, the load on this node and the possible resulting latency may be mitigated by using a tree structure to gather the information, making the load and latency proportional to logp. We expect this mechanism to remain sufficiently efficient for very large clusters.

Unfortunately, while IB seldom loses packets, multicast is basically an unreliable service. Thus, to guarantee reliability, we rely on BSP's synchronous nature to implement a Nack-based protocol as follows. If a node does not receive an information vector after a predefined time has elapsed since reaching a barrier, it explicitly requests the vector from the manager node using a normal (reliable) message. The timeout is set such that differences in barrier arrival times at different nodes do not result in unnecessary vector requests. If the manager node receives a vector request before completing the barrier, it responds immediately. Otherwise, vector requests are detected and handled in the next barrier synchronization. (If a node has not received an information vector for a certain barrier, it cannot proceed to the next superstep and send an information message corresponding to the next barrier. In this case, the manager node cannot pass the next barrier before handling the vector request.) The foregoing scheme does not send Nack control messages in the common case. In contrast, achieving reliable hardware multicast using Ack-based protocols requires sophisticated Ack aggregation techniques [15, 16].

- **0-copy receives.** In order to reduce memory copies at the receiver, we transfer data directly into its final remote memory location. This requires relevant remote memory regions to be registered with IB. Luckily, the BSP registration APIs can be used to identify such regions exactly [11]. However, for fine-grain data access, transferring the data of each `bsp_put` using a separate DTO introduces considerable overhead and loses the advantages of combining messages. Thus, we use IB's capability to scatter a chunk of incoming data into multiple memory locations.

  Since IB does not support remote scatter lists (a DTO can only describe local scatter/gather buffers), this can be achieved in two ways: (1) fetching and scattering a chunk of data by posting an RDMA-Read DTO with a scatter list at the receiver; (2) scattering an incoming chunk (sent by a Send DTO) by posting a receive DTO with a scatter list. In both cases, the sender must supply the receiver the placement information before the data transfer takes place. We have chosen the first option to facilitate flow-control. During registrations, nodes exchange LMR information. This

enables the sender to prepare control headers directly in DTO descriptor format that is suitable for the receiver; the DTO data is packed separately in continuous buffers. The receiver merely posts the readily available descriptors to the hardware and pulls chunks according to a congestion avoidance schedule similar to the baseline implementation. Once the receiver has successfully pulled all expected data from a certain endpoint, it sends a short conformation message to its peer, informing it that it is safe to reclaim the source DTO buffers. Thus, using RDMA incurs an additional round trip latency (accounting for the RDMA read and confirmation message) compared to the baseline scheme.

4 Performance Evaluation

In this section we evaluate the performance of our implementation. All experiments were performed on a cluster of 16 Xeon 3.2 GHz machines with 1GB memory and 1MB L2 cache, running Linux 2.6.4 and the Voltaire [17] uDAPL/IB stack. The machines were interconnected in a star topology using a 10Gbs InfiniBand SAN. Specifically, each machine employed a Mellanox [18] InfiniHost IIIEx MT25208 Host Channel Adaptor (HCA), which was plugged in an 8x PCI Express slot. We report results both for the resulting BSP machine parameters and application speedups.

4.1 Micro-benchmarks and BSP machine parameters

To measure our cluster's basic performance characteristics as well as the resulting BSP architectural parameters, we used the bsp_bench program from the BSPedupack distribution [19]. Given the number of CPUs to run on, bsp_bench measures the average FLOP computing rate by performing a DAXPY operation (double precision A times X plus Y, where A and Y are vectors) on 1000 double-precision (8 bytes) vectors. The l and g architectural parameters are measured by plotting the execution time of communication-only supersteps for a wide range of h-relations versus the corresponding values of h. The plots are approximated by a line; g is the slope of this line, and l is its y-intercept. Consequently, l actually denotes the latencies incurred by both the BSP implementation and the network lumped together, and g denotes the asymptotic time for transferring a single word to its final destination in memory in continuous traffic, accounting for the bsp_put calls, the data transfer, and the data placement. bsp_bench is a conservative benchmark: it times full h-relations, i.e., every processor sends and receives h data words; it uses bsp_put rather than bsp_hpput; and finally, all communication is generated using fine-grained bsp_put calls of a singe double-precision word each. Unless noted otherwise, all measurements were conducted with h-relations containing between 0 (the empty relation) and 4K.

In Table 1, we depict the BSP parameters for 8 nodes with the baseline implementation (BSP-IB-B) as well as the header compression (BSP-IB-H), multicast (BSP-IB-M), and RDMA (BSP-IB-R) optimizations. The l and g parameters are also given in absolute units for a qualitative comparison. As a reference, we also include the parameters of BSP machines

Table 1: BSP architectural parameters.

Architecture	r [Mflops/s]	g [flops/word]	l [flops]	g [μs/word]	l [μs]
BSP-IB-B	840	179	44K	0.21	52
BSP-IB-H	840	160	44K	0.19	52
BSP-IB-M	840	184	43K	0.21	51
BSP-IB-R	840	262	97K	0.31	115
IBM RS/6000SP	212	187	148K	0.88	698
SGI Origin 2000	326	297	95K	0.91	294
SGI Origin 3800	285	126	32K	0.44	114
Cray T3E	35	31	1K	0.88	34

based on the state-of-the-art Oxford BSP Toolkit [20] for the Cray T3E, SGI Origin 2000, SGI Origin 3800, and IBM RS/6000 SP multiprocessors, taken from [4]. These parameters were also obtained on 8 CPUs, using exactly the same benchmark code.

While the Cray is clearly the most "balanced" machine, BSP-IB is superior to BSP machines based on SGI Origin 2000 and IBM SP in every parameter. Compared to Origin 3800, BSP-IB has slightly higher (normalized) l and g values. This shows that the BSP model efficiently exploits the benefits of modern interconnects. In other words, BSP enables clusters to achieve a balanced architecture that matches those of expensive special purpose machines. When comparing raw performance, our high-end cluster improves on these widely used machines in terms of CPU speed, communication bandwidth, and latency by factors of 2-4. In fact, the cluster is an order of magnitude faster (per processor) than the Cray T3E supercomputer, has four times the bandwidth, and incurs a comparable latency.

The contributions of our further optimizations beyond BSP-IB-B are as follows. BSP-IB-H improves g by 10%. However, BSP-IB-M's improvements are barely noticed on 8 nodes, and BSP-IB-R increases both g and l considerably. The degradation in g can be attributed to the fine-grained access patterns, which force multiple I/O requests due to our hardware's limitation of 28 scatter-gather entries per DTO; the higher l is due to the additional descriptor manipulation and round trip time incurred by the RDMA implementation.

To better determine the benefits of BSP-IB-M, we ran bsp_bench on different numbers of nodes ranging from 1 to 16. We also included for reference a modified version of the benchmark, bsp_sparse, which implements sparse h-relations, i.e., node i sends data only to node i+1 (mod p). Sparse relations are common in many application communication patterns. As can be seen in Figure 2(a-b), in both benchmarks, BSP-IB-M becomes dominant compared to BSP-IB only at 16 nodes, with more pronounced improvements in bsp_bench. As can be expected, sparse-relations incur less latency than full-relations.

BSP-IB-R is naturally beneficial for large h-relations and coarse-grained data-transfers, which offset its increased setup latency and higher number of I/O operations. Thus, we experimented with varying words per bsp_put operation for h-relations up to 32K (on 8 nodes). We also plotted the performance of a variation of BSP-IB-R that does not use

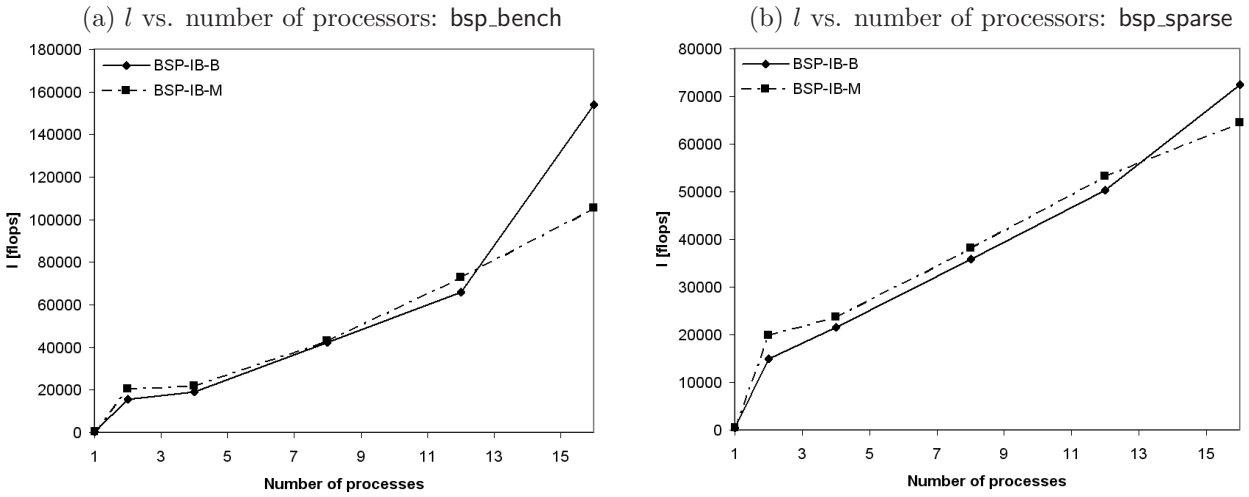

Figure 2: Impact of hardware multicast on latency for full and sparse relations.

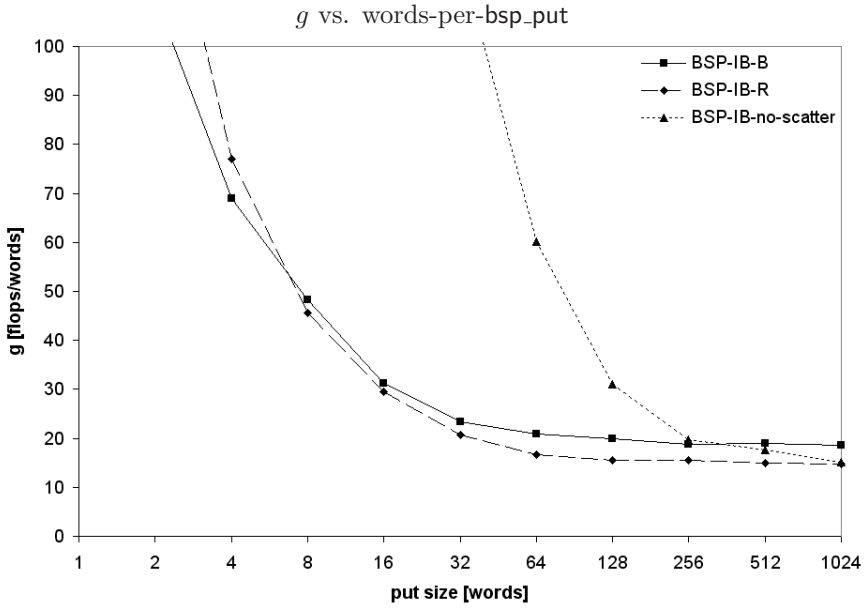

Figure 3: Impact of bsp_put granularity on BSP-IB, BSP-IB-R, and BSP-IB-no-scatter for up to 32K-relations.

scatter-lists at all, BSP-IB-no-scatter, to evaluate their contribution. The corresponding g measurements are presented in Figure 3. BSP-IB-R outperforms (lower g) BSP-IB starting at as little as 8 words per bsp_put. In contrast, the crossover of BSP-IB-no-scatter and BSP-IB takes place only at 512 words. These results highlight the importance of hardware scatter-gather capabilities for scientific computations.

Next, we validated the g measurements for extremely large relations, i.e., up to a 1M-relation in which every node sends and receives 8MB, using both fine-grain (single word per bsp_put) and bulk (single bsp_put per endpoint) relations. Unfortunately, due to an unsupported feature in the network hardware, when the receiver temporarily runs out of free buffers, the sending HCA backs off for a second before attempting to resume transmission, even though free receive buffers become available much earlier. Thus, in fine-grain relations, we had to post an amount of 16KB receive buffers that accommodates all incoming data in order to get meaningful results. Interestingly, this problem does not occur for coarse-grain relations because the CPU keeps pace with the IO (so free receive buffers are always available). Fine-grain relations resulted in $g = 197$ for BSP-IB and $g = 267$ for BSP-IB-R, incurring a small performance penalty compared to small relations. Bulk-relations resulted in $g = 21$ for BSP-IB and $g = 17$ for BSP-IB-R, similar to the results in Figure 3.

We conclude that in terms of g, BSP-IB-H is beneficial for fine-grained relations, and BSP-IB-R, for coarse-grain relations. However, as BSP-IB-R transfers control information in DTO descriptor format, it does not allow header compression. Thus, a dynamic scheme that transfers fine-grain data using header-compression and coarse-grain data using RDMA is required. In terms of l, BSP-IB-M is beneficial for more than 16 nodes for any communication pattern, and should thus be applied unconditionally in large clusters. BSP-IB-R, in contrast, has a negative impact on l. Nevertheless, large relations offset it, so a dynamic scheme is also advisable here. Such dynamic policies are an interesting subject for future work.

4.2 Application Speedups

We measured the speedup of BSP-IB for three BSPedupack [19] benchmark applications: Inner-product (IP), LU-decomposition (LU), and one-dimensional FFT (FFT), and compared their performance to that reported in [4] for an SGI Origin 3800 BSP multiprocessor. In all applications, the data is assumed to be properly distributed before the computation commences. Thus, the IP benchmark actually measures relative execution times of local IP computation (on $1/p$ of the given dataset size) and global sum reduction. Both in LU and in FFT, all auxiliary data structures are assumed to be pre-allocated and registered with BSP. The results are presented in Figures 4-6 (dataset sizes are detailed in the legend). BSP-IB offers consistently better speedups than the Origin 3800 for the same dataset sizes. Our optimizations had only mild effects on the results. Thus, to evaluate their impact on parallel applications, additional benchmarks that demonstrate more demanding communication patterns are required, and are a subject for future work. Finally, we note that initial comparisons between BSP-IB and MPI on our clusters have revealed similar results, suggesting BSP as an attractive alternative programming model for clusters in view of its simplicity from a programmer perspective. We plan to continue research of the relative strengths and weaknesses of these models in the future.

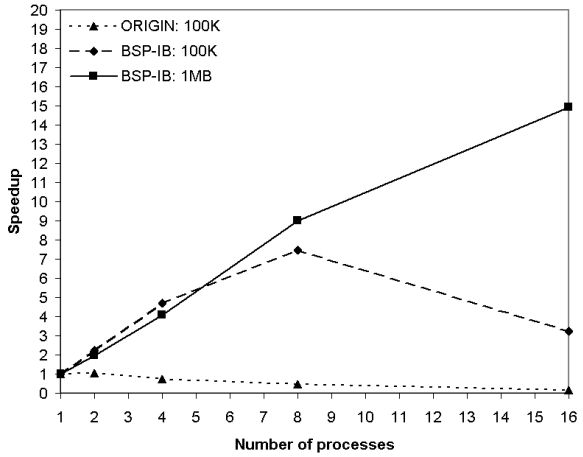

Figure 4: IP speedups.

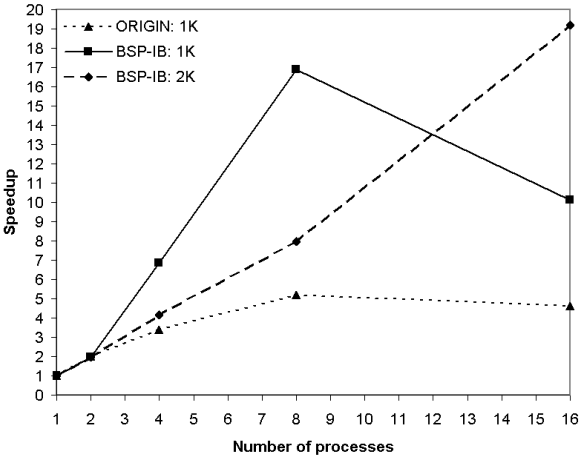

Figure 5: LU speedups.

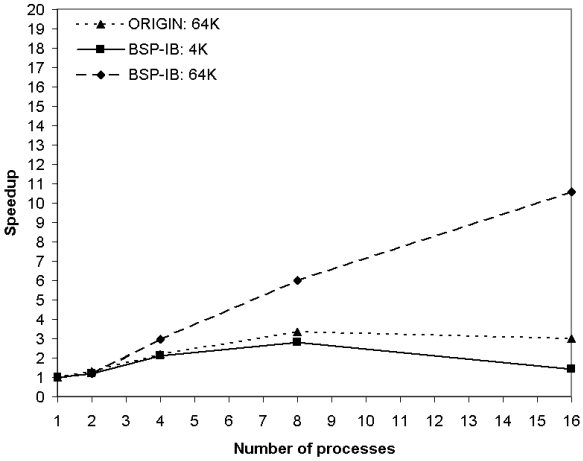

Figure 6: FFT speedups.

5 Conclusions

We have designed and implemented a high-performance BSP machine based on commodity hardware, which delivers comparable performance and a similarly balanced architecture as BSP machines based on expensive special-purpose multiprocessors. This was attained through extensive exploitation of the capabilities of InfiniBand and its natural match to the needs of BSP. The implementation revealed several novel insights for HPC and IB based computing clusters: scatter lists, which were designed mainly for storage buffers and protocol encapsulations, can also contribute substantially to HPC, but their permissible size must be increased in order to do so; 0-copy data transfers, which were designed primarily as a vehicle to free CPU cycles, can also improve data-placement bandwidth even when the CPU can be dedicated completely to this task; and, in synchronous systems, using Nack-based protocols instead of Ack-based ones can be used to guarantee the reliability of unreliable services without introducing overheads in the common case. As a final note, the simplicity of the BSP model and its opportunities for efficient implementations on contemporary SAN-based clusters may open the door for "shared-memory" style programming on large-scale clusters that deliver predictable, high performance.

References

[1] "Mpi: A message passing interface standard," *International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, pp. 165–414, 1994.

[2] L. Valiant, "Bulk synchronization parallel computers," *Parallel Processing and Artificial Intelligence*, pp. 15–22, 1989.

[3] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling, "Bsplib: the bsp programming library," *Parallel Computing*, vol. 24, pp. 1947–1980, 1998.

[4] R. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004.

[5] D. Skillicorn, J. Hill, and W. McColl, "Questions and answers about bsp," *Scientific Programming*, vol. 6, no. 3, pp. 187–207, 1997.

[6] J. Hill, S. Donaldson, and A. McEwan, "Installation and user guide for the oxford bsp toolset (v1.4) implementation of bsplib," *Technical Report, Oxford University Computing Laboratory*, 1998.

[7] O. Bonorden, B. Juurlink, I. V. Otte, and I. Rieping, "The paderborn university bsp (pub) library," *Parallel Computing*, vol. 29, pp. 187–207, 1999.

[8] Quadrics, `http://www.quadrics.com/`.

[9] Myrinet, http://www.myri.com/.

[10] InfiniBand Trade Association - InfiniBand Specification, http://www.infinibandta.com/.

[11] Y. Kee and S. Ha, "xbsp: an efficient bsp implementation for clan," in *Proceedings of Cluster Computing and the Grid*, 2001.

[12] User-Level Direct Access Transport APIs (uDAPL), Dat Collaborative, http://www.datcollaborative.org/.

[13] J. Hill and D. Skillicorn, "Lessons learned from implementing bsp," *Future Generation Computer Systems*, vol. 13, March 1998.

[14] J. Liu, A. Mamidala, A. Vishnu, and D. K. Panda, "Performance evaluation of infiniband with pci express," in *Proceedings of Hot Interconnect (HOTI)*, August 2004.

[15] J. Liu, A. Mamidala, and D. K. Panda, "Fast and scalable mpi-level broadcast using infiniband's hardware multicast support," in *In Int'l Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.

[16] A. Mamidala, J. Liu, and D. K. Panda, "Efficient barrier and allreduce on iba clusters using hardware multicast and adaptive algorithms," *IEEE Cluster Computing*, September 2004.

[17] Voltaire, http://www.voltaire.com/.

[18] Mellanox Technologies, http://www.mellanox.com/.

[19] BSP EDUpack distribution, available from http://www.bsp-worldwide.org/.

[20] Oxford BSPToolkit, available from http://www.bsp-worldwide.org/implmnts/oxtool/.