Optimization of Collective Communication in Intra-Cell MPI

M. K. Velamati[1], A. Kumar[1], N. Jayam[1], G. Senthilkumar[1], P.K. Baruah[1], R. Sharma[1], S. Kapoor[2], and A. Srinivasan[3]

[1]Dept. of Mathematics and Computer Science, Sri Sathya Sai University.

[2]IBM, Austin.

[3]Dept. of Computer Science, Florida State University.

Abstract: *The Cell is a heterogeneous multi-core processor, which has eight co-processors, called SPEs. The SPEs can access a common shared main memory through DMA, and each SPE can directly operate on a small distinct local store. An MPI implementation can use each SPE as if it were a node for an MPI process. In this paper, we discuss the efficient implementation of collective communication operations for intra-Cell MPI, both for cores on a single chip, and for a Cell blade. While we have implemented all the collective operations, we describe in detail the following:* barrier, broadcast, *and* reduce. *The main contributions of this work are (i) describing our implementation, which achieves low latencies and high bandwidths using the unique features of the Cell, and (ii) comparing different algorithms, and evaluating the influence of the architectural features of the Cell processor on their effectiveness.*

1. Introduction

The Cell is a heterogeneous multi-core processor from Sony, Toshiba and IBM. It consists of a PowerPC core (PPE), which acts as the controller for eight SIMD cores called Synergistic Processing Elements (SPEs). The SPEs are meant to handle the bulk of the computational load, but have limited functionality and local memory. On the other hand, they are very effective for arithmetic, having a combined peak speed of 204.8 Gflop/s in single precision and 14.64 Gflop/s in double precision. Furthermore, two Cell processors can be connected by the Cell Broadband Engine Interface to create Cell blade with a global shared memory.

There has been much interest in using it for High Performance Computing, due to the high flop rates it provides. However, applications need significant changes to fully exploit the novel architecture. A few different models of the use of MPI on the Cell have been proposed to deal with the programming difficulty, as explained later. In all these, it is necessary to implement collective communication operations efficiently within each Cell processor or blade.

In this paper, we describe the efficient implementation of a variety of algorithms for a few important collective communication operations, and evaluate their performance. The outline of the rest of the paper is as follows. In §2, we describe the architectural features of Cell that are relevant to the MPI implementation. We then describe MPI based programming models for the Cell in §3. We explain common features of our implementations in §4. We then describe the implementations and evaluate the performance of MPI_Barrier, MPI_Broadcast, and MPI_Reduce in §5, §6, and §7 respectively. We summarize our conclusions in §8.

2. Cell Architecture and Programming Environment

Figure 1 provides an overview of Cell processor. It consists of a cache coherent PowerPC core and eight SPEs running at 3.2 GHz, all of whom execute instructions in-order. It has a 512 MB to 2 GB external main memory, and an XDR memory controller provides access to it at a rate of 25.6 GB/s. The PPE, SPE, DRAM controller, and I/O controllers are all connected via four data rings, collectively known as the EIB. Multiple data transfers can be in process concurrently on the EIB – up to 128 outstanding DMA requests between main storage and the SPEs. The EIB's maximum bandwidth is 204.8 GB/s. The Cell Broadband Engine Interface (BEI) manages data transfers between the EIB and I/O devices. One of its channel can be used to connect to another Cell processor at 25.6 GB/s, creating a Cell blade with a logical global shared memory.

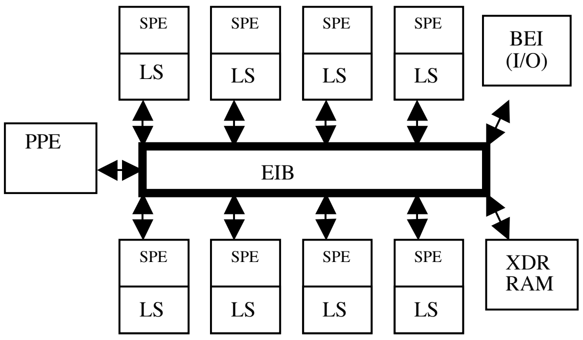
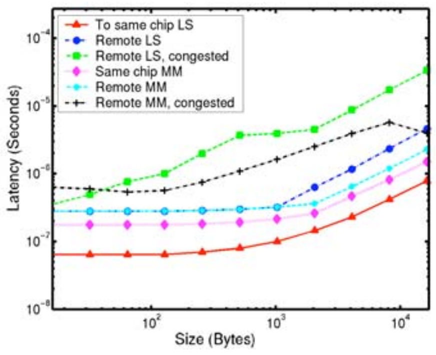


Figure 1: Overview of the Cell processor.

Each SPE has its own 256 KB local memory from which it fetches code and reads and writes data. Access latency to and from local store is 6 cycles. *All loads and stores issued from the SPE can only access the SPE's local memory. Any data needed by the SPE that is present in the main memory must be moved into the local store explicitly, in software, through a DMA operation.* An SPE can have up to sixteen outstanding requests in its DMA queue. The maximum DMA size is 16 KB.

The DMA commands may be executed out-of-order. Partial ordering can be ensured by fenced or barriered DMAs. The former is executed only after all previously issued DMAs with the same tag on the same SPE DMA queue have completed. The latter has the same guarantee, but also ensures that all subsequent DMAs issued on the same DMA queue with the same tag execute after it has completed. A DMA list command can be used to scatter data or gather data to or from multiple locations respectively. It occupies only one slot in the DMA queue.

In order to use the SPEs, a process running on the PPE can spawn a thread that runs on the SPE. Each SPE can run one thread, and the thread accesses data from its local store. The SPE's local store and registers are mapped onto the effective address of the process that spawned the SPE thread. Data can be transferred from the local store or register of one SPE to that of another SPE by obtaining this memory mapped address, and performing a DMA. Similarly, SPE threads can DMA data to or from main memory locations allocated by the spawning thread. The SPEs are capable of limited functionality. So, for each SPE thread, a proxy thread is created on the PPE which handles requests for the corresponding SPE thread, such as system calls. It is, therefore, advisable to limit the use of the PPE, since it is involved with other coordination tasks.



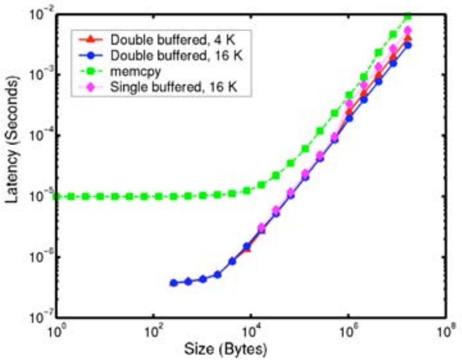


Figure 2: *Left:* Latency for DMA *put* operation (*get* timings are similar) issued by an SPE to local store (LS) or main memory (MM). *Right:* Latency for memory to memory copy through an SPE with different buffer sizes on local store, or by the PPE using *memcpy*.

We observe the following regarding the performance of DMAs, some of which are presented in figure 2 (left): (i) SPE-SPE DMAs are much faster than SPE-main memory DMAs on the same chip, (ii) sending multiple smaller DMAs is slower than sending fewer long ones from the same SPE, (iii) latency between SPEs on different chips are significantly higher than those on the same chip, (iv) maximum bandwidth between SPE and main memory is around 7 GB/s, while between SPE and SPE it is around 25 GB/s, (v) latency is higher in the presence of congestion, when multiple SPEs are transferring data, and (vi) the variance of the latency is higher with congestion.

3. MPI Based Cell Programming Models

The heterogeneous architecture and the small local stores of the SPEs make programming the Cell difficult. Some of the programming models to deal with this challenge are based on MPI. In the MPI microkernel model [7], the application is divided into several smaller tasks with small code and data size. A scheduler schedules the tasks on the SPEs. In another model [3], an existing application is ported to the Cell by treating each SPE as if it were a node for an MPI process, using the main memory to store the application data, and the local store as software controlled cache. The software cache feature of SDK 2.0 can help with the last aspect. Large code size can be dealt with by bringing in the code as needed through code overlaying. SDK 2.0, again, facilitate this. *This is the model for which we target our MPI implementation, assuming that application data is in main memory, and that the MPI calls are provided the effective addresses of these locations, where they require pointers to memory locations.* If the application data is in local store, then more efficient implementations can be developed. *We also discuss only the case of contiguous data.* Use of non-contiguous data will lead to higher latencies. Note that this implementation can also be helpful with clusters of Cell processors or blades – this implementation can be used for the portion of communication that happens within a chip or blade, which is combined with MPI communication connecting different Cell processors or blades. Similar strategies have been developed for SMP clusters on other hardware [5, 9, 11].

4. Common features of Intra-Cell Collectives

We will call each SPE thread as a process, and an SPE as a processor, for the sake of consistency with usual usage in MPI. Let P be the number of MPI processes. Each SPE runs one thread at most, and so P SPEs are involved in the computation. Each SPE maintains a metadata array of P elements in its local store. (This memory usage can be decreased for some algorithms.) Each entry is 16 Bytes; smaller space would suffice, but this size is forced by DMA alignment requirements. With the maximum of 16 SPEs on a blade, this requires 256 B, which is small. The barrier call has a separate metadata array to avoid interference with other calls. The implementation also allocates two buffers of 16 KB each on the local store to use as software controlled cache. Timing results, such as in figure 2 (right), indicate that buffers of 4 KB each would yield comparable performance. The implementation tries to minimize data transfers involving the main memory, because of the larger latencies involved in such transfers, compared with that to local store on-chip. The bandwidth to main memory is also the bottleneck to most algorithms, and thus access to it should be minimized. The two buffers above are used instead; the use of multiple buffers helps reduce latency by enabling double buffering – when one buffer has data being transferred out of it, another buffer is used to transfer data into the SPE.

SPE i typically transfers data to SPE j by DMAing data to metadata array location i on SPE j. SPE j polls this entry, and then DMAs data from a local store buffer on SPE i to one on SPE j. It then typically acknowledges receipt to SPE *i* by DMAing to metadata entry *j* on SPE *i*. Serial numbers are often used in the metadata entries to deal correctly with multiple transfers. Writing to a metadata entry is atomic, because DMAs of size up to 128 B are atomic. However, the DMAs may be executed out of order, and the data sent may also differ from the data at the time the DMA was queued, if that location was updated in the meantime. We don't discuss the implementation details to ensure correctness in the presence of these issues, in order to present a clearer high level view of the algorithms. In order to simplify some of the implementation, we made the collective calls synchronize at the end of each call, using a barrier. We later show that the barrier implementation on the Cell is very efficient.

The experimental platform was a Cell IBM QS20 revision 5.1 blade at Georgia Tech, running Linux. The *xlc* compiler for the Cell, with optimization flag *–O5*, was used. We did not have dedicated access to this system. However, timings for some of the algorithms on a Cell blade at IBM Rochester, to which we had dedicated access, yielded similar results. The timings were performed using the decrementer register on the Cell. This has a resolution of around 70 nano-seconds. The variances of the timing results for collective calls, other than the barrier, were fairly small. The variance for the barrier, however, was somewhat higher.

5. Barrier

This call blocks the calling process until all the other members of the group have also called it. It can return at any process only after all the group members have entered the call.

5.1 Algorithms: We have implemented three classes of algorithms, with a few variants in one of them.

Gather/Broadcast: In this class of algorithms, one special process, which we call the root, waits to be informed that all the processes have entered the barrier. It then broadcasts this information to all processes. On receiving the information broadcast, a process can exit the barrier. We have implemented the following three algorithms based on this idea. We give an abbreviation for each algorithm along with its name, which will be used to refer to it later.

(OTA) One-to-all: Here, an SPE informs the root about its arrival by setting a flag on a metadata entry in the root. The root waits for all its entries to have their flag set, unsets these flags, and then sets a flag on a metadata entry of each SPE. These SPEs poll for this flag to be set, then unset it and exit. Note that polling is quite fast because the metadata entry is in the local store for each SPE performing the polling; the bottlenecks are (i) DMA latency, and (ii) processes arriving late. The broadcast phase of this algorithm, where the root sets flags, has two variants. In the first one, the root uses a DMA *put* to transfer data to each SPE. An SPE can have sixteen entries in its own DMA queue, and so can post the DMA commands without blocking. In the second variant, the root issues a single *putl DMA List* command.

Note: In the description of subsequent algorithms, we shall not describe the re-initialization or unsetting of variables or flags, in order to present a clearer high-level view of the algorithms.

(SIG) Use a signal register: The signal registers on each SPE support one-to-many semantics, whereby data DMAed by an SPE is *OR*ed with the current value. The broadcast phase of this algorithm is as in OTA. The gather phase differs in that each SPE sets a different bit of a signal register in the root, and the root waits for all signals to be received.

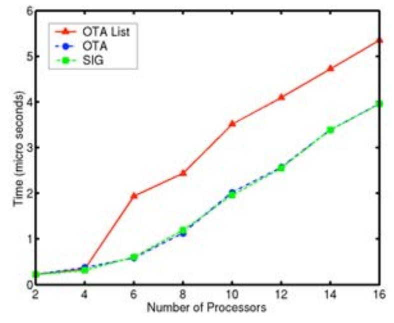
(TREE) Tree: This gathers and broadcasts data using the usual tree based algorithm [12]. In the broadcast phase of a binomial tree algorithm, the root starts by setting a metadata flag on another

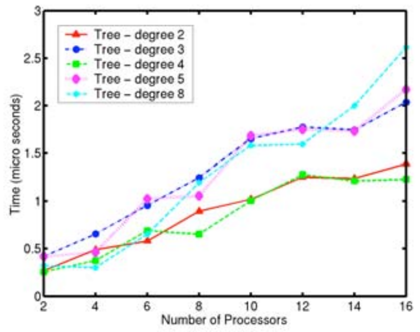
SPE. In each subsequent phase, each process that has its flag set in turn sets the flag of one other SPE. Thus, after i phases, 2^i processes have their flags set. Therefore $\lceil \log_2 P \rceil$ phases are executed for P SPEs. In a tree of **degree *k*** [1, 12], in each phase SPEs, which have their flag set, set the flags of $k - 1$ other distinct SPEs, leading to $\lceil \log_k P \rceil$ phases. The gather step is similar to that in the broadcast, but with the directions reversed.

Pairwise-exchange (PE): This is a commonly used algorithm for barriers [12]. If P is a power of 2, then we can conceptually think of the SPEs as organized as a hypercube. In each phase, an SPE exchanges messages with its neighbor along a specific dimension. This exchange is performed by a process setting a flag in it neighbor using DMA, and then waiting for the neighbor to set its flag. The barrier is complete in $\log_2 P$ phases. If P is not a power of two, then a slight modification to this algorithm [12] takes $2 + \lfloor \log_2 P \rfloor$ steps.

Dissemination (DIS): This is another commonly used algorithm for barriers [12]. In the i th phase here, SPE j sets a flag on SPE $j+2^i \pmod P$ and waits for its flag to be set by SPE $P+j-2^i \pmod P$. This algorithm takes $\lceil \log_2 P \rceil$ steps, even if P is not a power of two.

5.2 Performance Evaluation: We next evaluate the performance of the above algorithms.





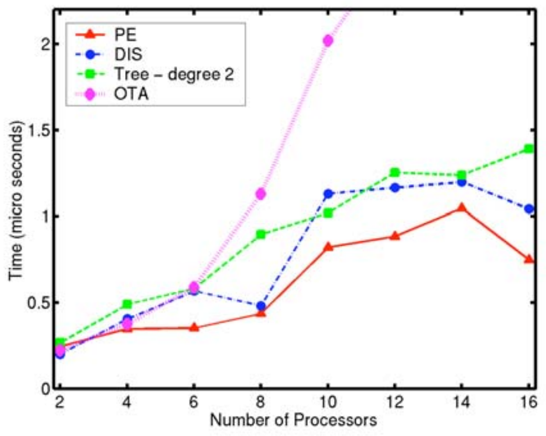


Figure 3: Barrier latencies. *Top left:* Comparison of OTA with and without DMA list, and SIG without DMA list. *Bottom left:* Comparison of TREE with different degrees. *Right:* Comparison of four barrier algorithms.

We first see that the use of DMA lists does not improve the performance of the barrier – in fact, the performance is worse when P is greater than four. We also see that the use of the signal register in the gather phase does not improve performance compared with the use of plain DMAs, which are used in OTA.

We next evaluate the influence of tree degree in the TREE algorithm. In this algorithm, we make slight optimizations when the tree degree is a power of 2, replacing modulo operations with bit-wise operations. This difference is not sufficient to explain the large difference in times seen for degree 2 and 4, compared with other degrees. We believe that the compiler is able to optimize a for loop involved in the computation better with power of two degrees. However, increasing the degree to eight lowers the performance. This can be explained as follows. As the tree degree increases, the number of phases decreases. However, the number of DMA issued by the root increases. Even though it can queue up to sixteen DMAs, and the messages sizes are small enough that the bandwidth is not a limiting factor, each DMA from the queue has to wait for its turn. Consequently, having multiple DMAs in the queue can lower the performance. This trend is also shown by DMA timing results, which are not presented here.

The PE and DIS algorithms perform substantially better than the gather/broadcast type of algorithms, with PE being clearly better than DIS when *P* is greater than eight. Before explaining this, we first discuss a factor that sometimes influences the performance of DIS. In contrast to PE, where pairs of processes exchange information, in DIS, each process sends and receives messages to different processes. On some networks, exchange between processes is faster than sending and receiving between different processes, which can cause DIS to be slower than PE. This is not the case here. DMA tests show that exchanging data is no faster than communication between different SPEs. The reason for the difference is that when the number of processes is greater than eight, some of the processes are on a different chip. The DMA latency between these is higher. In PE, all the inter-chip DMAs occur in the same phase. In DIS, this occurs in each phase. Thus each phase gets slower, where as in PE, only one of the phases is slowed down due to this fact. This slower phase also explains the sudden jump in latency from eight to ten processes.

5.3 Alternatives: We have also considered two of alternate algorithms. An algorithm based on atomic increments has been proposed for SMPs [9]. Here, each process increments a shared variable and waits for the variable to attain a value *P*. The performance is not good in conventional SMPs because of excessive cache misses. This algorithm is not effective on the Cell either, because atomic increments can be performed only on main memory, and not on the local stores of other SPEs, incurring a higher latency. In fact, the performance is much worse than that of DMAs to main memory, and increments take the order of microseconds, when several SPE simultaneously attempt an atomic increment, even though they take well under a micro second when only one SPE performs them. Thus this scheme is not competitive with those presented above. Another alternative is to have the PPE coordinate the synchronization. SPEs and the PPE can communicate short messages of 32 bits using a *mailbox*. Once the PPE receives a mailbox message from each SPE, it can send one back to each SPE, indicating that it can exit the barrier. However, this takes tens of microseconds when multiple SPEs have to be handled, and is not competitive with the algorithms presented above.

5.4 Related work: Related work can be classified along the following lines: (i) Implementations on top of Send/Receives, (ii) RDMA based implementations, and (iii) SMP based implementations. The TREE, DIS, and PE algorithms can be implemented on all the three types of implementations. Atomic increment for SMPs is described in [9]. Multicast based RDMA implementations are reported in [2, 6], which are like OTA, except that in the broadcast phase, the root does a hardware multicast to all the processes.

We present some results on other architectures in table 1. The barrier times on the Cell are much smaller, (i) partly because the hardware provides low latencies between local stores on the same chip, and the latency off-chip too is fairly small, and (ii) the implementation makes effective use of this by avoiding off-chip communication as far as possible, and avoids accessing main memory.

P	Cell (PE) μs	Myrinet[1] [12] μs	NEC SX-8[2] [8] μs	SGI Altix BX2[3] [8] μs
8	0.4	≈ 10	≈ 13	≈ 3
16	1.0	≈ 14	≈ 5	≈ 5

Table 1: Barrier times on different hardware.

6. Broadcast

In this call, the root broadcasts its data to all processes.

6.1 Algorithms: We discuss below five algorithms for broadcast.

(TREEMM) Send/Receive: This algorithm is the usual tree based Send/Receive algorithm [2, 10], with modifications given below. The tree structure is as in the broadcast phase of TREE for the barrier. However, instead of just setting a flag, a process that sends data also passes the main memory location of its application data. A receiving process copies this data to its own main memory location. This cannot be performed directly, because DMA is possible only between a local store address and an effective address. So, an SPE first copies a chunk of data from the source location to its local store, and then copies this back from the local store to the destination location in main memory. While this seems wasteful, a similar process occurs in regular cache-based processors, where copying a line can involve two or three cache misses. We ameliorate the DMA latency by double buffering. Performance tests on memory to memory copy in figure 2 (right) shows that double buffering yields a significant improvement in performance over single buffering. TREEMM's communication structure is similar to an implementation built on top of MPI_Send and MPI_Recv. However, it avoids the extra overheads of MPI calls by directly implementing the DMA calls in this routine. Furthermore, it avoids extra copy overheads, and uses double buffering to reduce the memory access latency (the latter is, in some sense, like prefetching to cache).

(OTA) Each SPU Copies its Data: In this implementation, the root broadcasts its metadata, as in the broadcast phase of barrier TREE. It sends the main memory location of the source data, in addition to setting the flag. Once an SPU receives this information, it copies data from the root's locations to its location in main memory, using double buffering. On some systems, simultaneous access to the same memory location can degrade performance by making this a hotspot. We include a **shift** S to avoid this. That is, SPE i first copies with an offset of $i \times S$, and then copies the initial portion. If $i \times S$ is greater than the data size, then this index wraps around.

(G) Root Copies All Data: In this implementation, the root gathers metadata from all processes in a tree structured manner. The metadata contains the destination addresses, in addition to the flag. The root then copies its data to each of the destination addresses. This is, again, done through double buffering. Each time data is brought in to local store, it is DMAed to all destination locations. With the maximum of sixteen SPEs possible, we need at most fifteen *put*s

[1] Each node here is a dual processor Xeon 2.4 GHz, and nodes are connected by Myrinet.

[2] Each node of the NEC SX-8 has eight vector processors capable of 16 Gflop/s, with 64 GB/s bandwidth to memory from each processor. The total bandwidth to memory for a node is 512 GB/s. The nodes are connected through a crossbar switch capable of 16 GB/s in each direction between each pair of nodes.

[3] The SGI Altix BX2 is a CC-NUMA system with a global shared memory. Each node contains eight Itanium 2 processors, and nodes are connected using NUMALINK4. The bandwidth between processors on a node is 3.2 GB/s, and between nodes 1.6 GB/s.

and one *get* pending in the DMA queue, and so the DMA requests can be placed in the queue without blocking.

(AG) Each Process Transfers a Piece of Data: In this implementation, all processes perform an *allgather* on their metadata to get the destination addresses of each process, and the source address of the root. Each process is then responsible for getting a different piece of data from the source and transferring it to the corresponding location in each destination. This is done in a double buffered manner, as with broadcast G. We also specify a **minimum size** for the piece of data any process can handle, because it may be preferable for a few processes to send large DMAs, than for many processes to send small DMAs, when the total data size is not very large. Increasing the minimum size decreases parallelism in the data transfer, with the potential benefit of fewer DMAs, for small messages.

(TREE) Local Store based Tree: In this implementation, the root gets a piece of data from main memory to its local store, and broadcasts this piece in a tree structured manner to the local stores of all processes. Each piece can be assigned an index, and the broadcast is done by having an SPE with data sending its children (in the tree) metadata containing the index of the latest piece that is available. A child issues a DMA to actually get this data. After receiving the data, the child acknowledges to the parent that the data has been received. Once all children have acknowledged receiving a particular piece, the parent is free to reuse that local store buffer to get another piece of data. A child also DMAs received data to its main memory location, and sends metadata to its children in the tree. In this implementation too, we use double buffering, so that a process can receive a piece into one buffer, while another piece is waiting to be transferred to its children. In this implementation, we denote **pipelined** communication between the local stores by a tree of **degree 1**.

6.2 Performance Evaluation: We first determine the optimal parameter for each algorithm, such as the tree degree, shift size, or minimum piece size. We have also evaluated the effect of other implementation choices, such as use of fenced DMAs and DMA lists, but do not discuss these here.

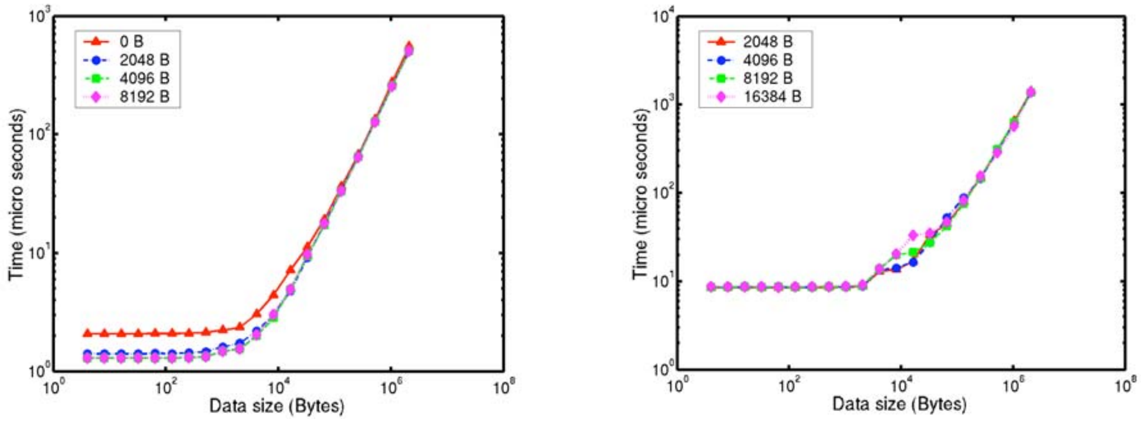


Figure 4: Broadcast timings. *Left:* OTA on four processors, with different shifts. *Right:* AG on sixteen processors, with different minimum sizes. Data for 2 KB and 4 KB are very close in both figures.

Figure 4 shows that on four processors, including some shift to avoid hotspots improves performance, though it is not very sensitive to the actual shift used. On larger numbers of processors, all shifts (including no shift) perform equally well. The likely reason for this is that with more processors, the time at which the DMA requests are executed varies more, and so we don't have a large number of requests arriving at the same time. Figure 4 also shows that on sixteen processors, using a minimum piece size of 2K or 4K in AG yields better performance than larger minimum sizes. (Lower sizes – 1KB and 128 B – perform worse for intermediate data sizes.) At small data sizes, there is only one SPE performing one DMA for all these sizes, and so performances are identical. For large data, all pieces are larger than the minimum, and so this size makes no difference. At intermediate sizes, the smaller number of DMAs does not compensate for the decrease in parallelism for minimum sizes larger than 4 KB. The same trend is observed with smaller numbers of processes.

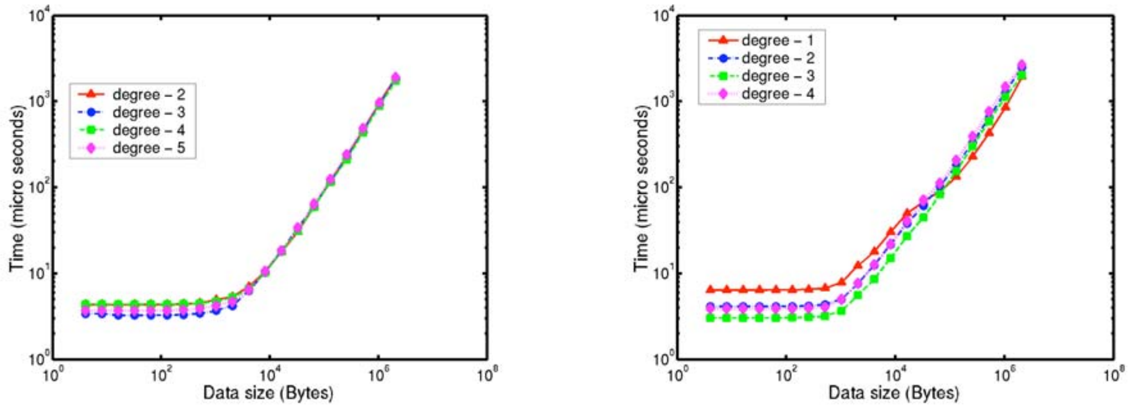


Figure 5: Broadcast timings. *Left:* TREEMM with different degrees on twelve processes. *Right:* TREE on sixteen processes. Data for degrees 2 and 4 are very close in both figures.

Figure 5 compares the effect of different tree degrees on the performance of the tree-based algorithms. In TREEMM, a tree degree of 3 yields either the best performance, or close to the best, for all process counts. A tree of degree two yields the worst performance, or close to that. However, the relative performances do not differ as much as they do in TREE. Furthermore, the differences show primarily for small messages. Note that for small messages, a higher tree degree lowers the height of the tree, but increases the number of metadata messages certain nodes send (unlike in a Send/Receive implementation, a parent sends only metadata to its children, and not the actual data). It appears that the larger number of messages affects the time more than the benefits gained in decrease of tree heights, beyond tree degree 3. A similar trend is demonstrated in TREE too, though the differences there are greater. For large messages, the performance of the different algorithms is similar, though the pipelined implementation is slightly better for very large data sizes. The latter observation is not surprising, because the time taken for the pipeline to fill is then negligible related to the total time, and the number of DMAs issued by any SPE subsequently is lowest for pipelining.

Figure 6 compares the performance of the different algorithms. The trend for the different algorithms on eight processes (not shown here) is similar to that on sixteen processes. We can see that AG has the best, or close to the best, performance for large messages. TREE degree 3 is best for small messages with more than four processes (its data points visually coincide with TREEMM degree 3 on twelve processes, and so are not very visible in that graph). Up to four processes, broadcast G is best for small messages. Pipelining is also good at large message lengths, and a few other algorithms perform well under specific parameters. As a **good choice of algorithms**, we use broadcast AG for data of size 8192 B or more, broadcast TR degree 3 for small data on more than four processes, and broadcast G from small data on four or fewer processes. The maximum bandwidth that can be served by the main memory controller is 25.6 GB/s. We can see that with this choice of algorithms, we reach close to the peak total bandwidth (for *P-1* writes and one read) for all process counts of four or more, with data size *16 KB* or more. The bandwidth per process can be obtained by dividing the total bandwidth by the number of processes.

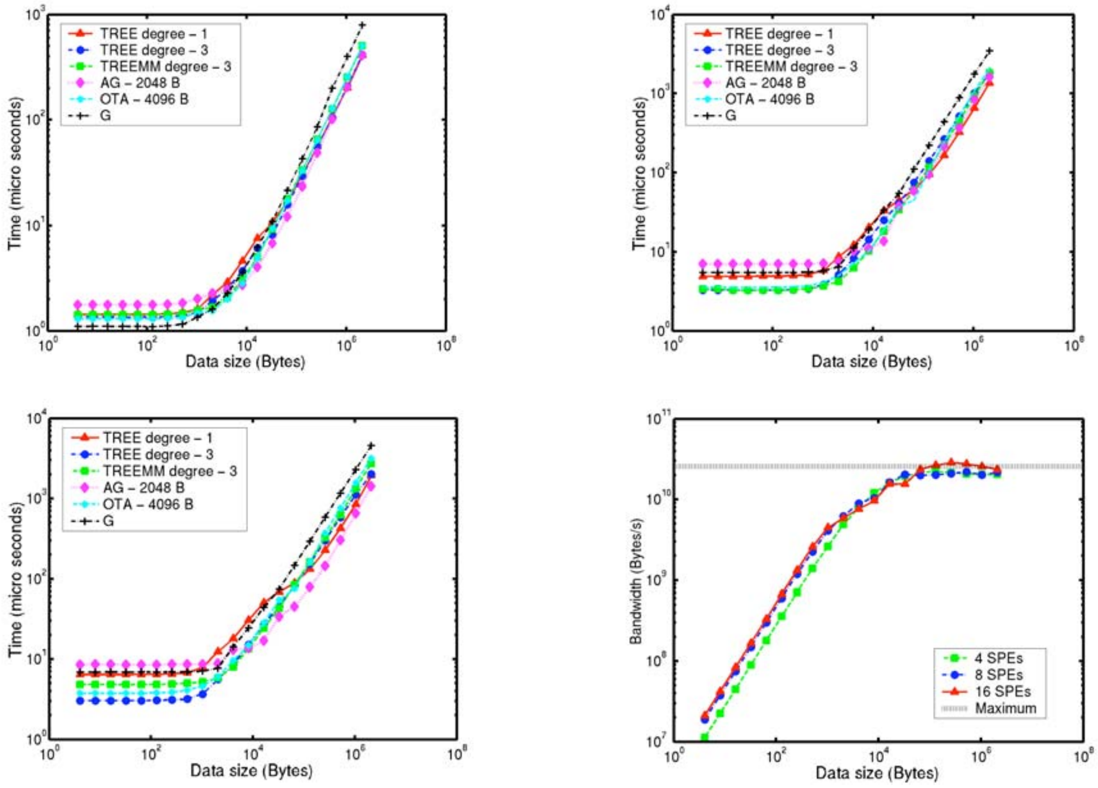


Figure 6: Broadcast performance. *Top left:* Timing for four processes. *Top right:* Timing on twelve processes. *Bottom left:* Timing on sixteen processes. *Bottom right:* Main memory bandwidth with a "good" choice of algorithms.

6.3 Alternatives: We also considered the alternative of having SPEs send their data locations to the PPE, and having PPE execute *memcpy* to transfer the data. However, *memcpy* on the PPE is much slower than transferring data using SPEs with double buffering, as shown in figure 2.

6.4 Related Work: Related work can be classified along the three lines mentioned in the barrier. Implementations of the TREEMM type algorithm, either on top of Send/Receive [10] or using RDMA features [1], have been reported in several studies. An OTA type broadcast has been studied on SMPs [9]. Send/Receive based implementations can also use van de Geijn's algorithm for large messages [10]. Here, the data is first scattered to all the processes, and then all-gathered. We have not directly implemented this, because a Send/Receive implementation is not the most effective on this architecture. However, broadcast AG uses this idea to a certain extent. RDMA multicast based implementations have also been reported [4, 5].

Table 2 compared the latencies on different hardware. The performance on the Cell is better than on other hardware, except for the NEC SX-8. The latter has memory bandwidth of 512 GB/s, which is about a factor of 20 more than on the Cell. However, the Cell implementation achieves a much greater fraction of the peak memory bandwidth possible. For example, the 1 MB eight-processor result yields useful memory bandwidth of 80 GB/s out of the 512 GB/s feasible on the SX-8, while the Cell results yield around 20 GB/s out of the 25 GB/s feasible.

Data Size	**Cell (PE) μs**		**Infiniband[4] μs**		**NEC SX-8** [8] **μs**		**SGI Altix BX2** [8] **μs**	
	P = 8	*P = 16*	*P = 8*	*P = 16*	*P = 8*	*P = 16*	*P = 8*	*P = 16*
128 B	1.7	3.1	≈ 18	≈ 10				
1 KB	2.0	3.7	≈ 25	≈ 20				
32 KB	12.8	33.7	≈ 220					
1 MB	414	653			≈ 100	≈ 215	≈ 2600	≈ 3100

Table 2: Broadcast times on different hardware.

7. Reduce

In this call, data from all the processes are combined using an associative operator, such as MPI_SUM for addition, and the result placed in the root. Two of the algorithms also assume that the operation is commutative, which is true for all the built-in operators.

7.1 Algorithms: The communication structure of this operation is similar to that of the broadcast, but with the directions reversed. In addition, each time a processor gets data, it also applies the operator to a current value and the new data. Since the communication structure is similar to the broadcast, we considered only the types of algorithms that worked well for the broadcast, namely, TREE and AG. Note that in both these algorithms, the computation can also be parallelized efficiently, unlike with OTA.

(TREE) Local Store based Tree: In this implementation, the communication direction of the broadcast TREE is reversed. A process gets a piece of data from its memory location to local store, gets data from a child's local store to its own local store when that data is available, and combines the two using the specified operator. It repeats this process for each child, except that it does not need to get its own data from main memory for subsequent children. Once it has dealt with all the children, it informs the parent about the availability of the data by DMAing a metadata entry, as in the broadcast. It repeats this for each piece of data in main memory. Double buffering is used to reduce the latency overhead by bringing data from main memory or the next child into a new buffer. Unlike with the broadcast, we need four buffers, two for each operand of a reduce operation, and two more because of double buffering. Due to space constraints on the local store, we used the same buffers as in the broadcast, but conceptually treated them as having half the size (four buffers of 8KB each instead of two buffer of 16K each with broadcast). Performance evaluation with four buffers of 16K each did not show much difference in performance.

[4] The Infiniband implementations used the RDMA multicast feature. The RDMA time on eight processors [5] uses a different implementation than the sixteen processor one [6], and so is slower. In the RDMA results, latencies for different implementation choices on different clusters are available. We chose the best results in each case considered.

(AG) Each Process Handles a Piece of Data: In this implementation, each process is responsible for reducing a different piece of the data, and then writing this to the destination location of the root. An initial all gather is used to get addresses of all SPEs, as in broadcast.

(TREEMM) Send/Receive: We implemented the usual tree based reduction algorithm on top of our implementation of MPI_Send and MPI_Recv [3] for the Cell processor, just for comparison purposes. The MPI_Send and MPI_Recv operations themselves make effective use of the features of the Cell. However, the extra copying to memory makes the performance slower.

7.2 Performance Evaluation: We give performance results in figure 7, for different numbers of processes. We can see that, for small data sizes, TREE degree 3 is either the best, or close to it, on greater than four processes. This is consistent with the behavior expected from the broadcast timings. On four processes, a tree of degree four, which has height 1, performs best. But, degrees 2, 3, and 4 are very close to each other in most cases. Reduce AG is worse for small messages, because of the overhead of all-gathering the metadata initially. TREE degree 1 is best for very large data, except on four processes. Reduce AG is the best at intermediate data sizes, except on four processes, where it is the best even for large messages. This can be explained as follows. Reduce AG parallelizes the computations perfectly, but issues $P+1$ DMAs for each piece of data (P gets and one put). As mentioned earlier, sending a large number of DMAs is less efficient. On four processes, this number is not very large, and so AG still performs well. The better performance of AG at intermediate data sizes can be explained as follows. The pipelined algorithm (TREE degree 1) requires communication of the order of the number of processes before results start coming out. If the data size is not very large, then this time plays a relatively large role. For large data, the initial time taken does not play as important a role, and so the pipelined algorithm is better. The reduction takes more time than the broadcast because of the extra overhead of the reduction operation. We have also evaluated the performance of the Send/Receive based reduce TREEMM. Its performance varies between *5 μs* for *128 B* to around *2000 μs* for *1 MB* on eight processors, and is worse than the best algorithm for each data size and processor count considered.

7.3 Related Work: Tree based algorithms have been implemented on top of Send/Receive [10] and on SMPs [9]. A reduce implementation based on shared Remote Memory Access (RMA) has also been reported [11]. An algorithm similar to AG has also been implemented on SMPs, though with extra copying involved [9]. Rabenseifner's algorithm has been implemented on top of Send/Receives for large messages. The idea is similar to van de Geijn's algorithm for broadcast in reducing the bandwidth associated cost. It performs a reduce-scatter followed by a gather at the root. Additional optimizations are described in [10].

We compare the performance of reduce on different hardware in table 3. For the Cell results, we used the best algorithm for each data size and processor count. Again, we see that the Cell performs very well, except in comparison with the SX-8, due to the much higher bandwidth available on the latter.

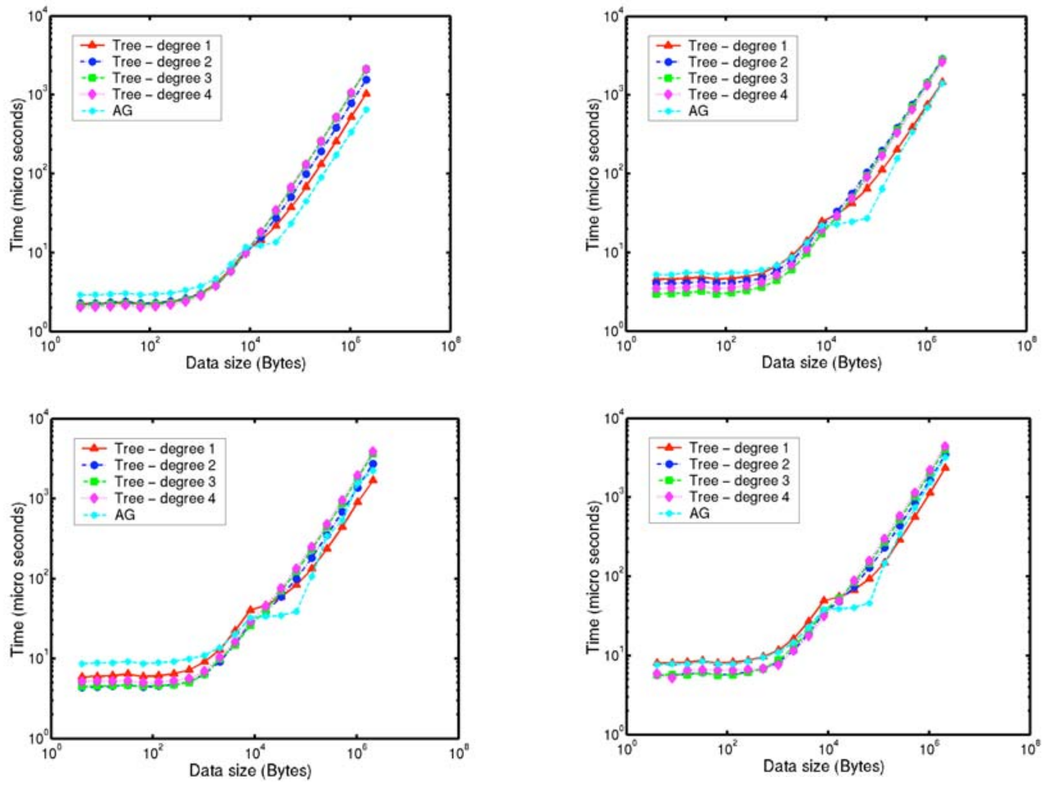


Figure 7: Reduce timing for MPI_SUM on MPI_INT (four Bytes per int). *Top left:* Four processes. *Top right:* Eight processes. *Bottom left:* Twelve processes. *Bottom right:* Sixteen processes.

Data Size	**Cell (PE)** µs		**IBM SP**[5] [11] µs		**NEC SX-8** [8] µs		**SGI Altix BX2** [8] µs	
	P = 8	*P = 16*	*P = 8*	*P = 16*	*P = 8*	*P = 16*	*P = 8*	*P = 16*
128 B	3.06	5.69		≈ 40				
1 KB	4.41	8.8		≈ 60				
1 MB	689	1129		≈ 13000	≈ 230	≈ 350	≈ 10000	≈ 12000

Table 3: Reduce times on different hardware.

8. Conclusions and Future Work

We have described in detail implementation of three MPI collective operations: *barrier*, *bcast*, and *reduce*. We have also implemented the following: *gather*, *allreduce*, *scan*, *allgather*, *alltoall*, and vector versions of these calls. Our results show good performance, both within a chip and in blade consisting of two Cell processors. We have implemented a variety of algorithms. While we use the availability of the common shared memory and high bandwidths available, the main benefit are obtained through effective use of the local store, and hiding the cost of access to main memory through double buffering.

Some of the future work is as follows. (i) Since the metadata size is sixteen bytes, we can consider using some of the extra space in it for small messages, thereby reducing the number of

[5] Each node of the IBM SP was a 16-processor SMP.

communication operations for small messages. (ii) We have performed barrier synchronization at the end of each collective call, in order to prevent successive calls from interfering with each other. This can be avoided by using counters, as some implementations have done, and may be beneficial when applications reach the collective call at much different times. (iii) It will be useful to consider the integration of this intra-Cell implementation with implementations that connect Cell blades using networks, such as Infiniband. (iv) If the application data is in local store, then our implementation can be made faster. This can be useful, for example, in the MPI microkernel model for programming the Cell.

Acknowledgements: We thank the Sony-Toshiba-IBM Cell Center of Competence at Georgia Tech for providing use of their IBM QS20 Cell blades, and IBM for providing access to their Cell blades under the VLP program. Most of all, we express our gratitude to Sri Sathya Sai Baba for bringing us all together to perform this work, and for inspiring and helping us toward our goal.

References

1. R. Gupta, P. Balaji, D.K. Panda, and J. Nieplocha, *Efficient Collective Operations Using Remote Memory Operations on VIA-Based Clusters*, Proceedings of IPDPS, 2003.
2. S.P. Kini, J. Liu, J. Wu, P. Wyckoff, and D.K. Panda, *Fast and Scalable Barrier Using RDMA and Multicast Mechanisms for Infiniband-Based Clusters*, Proceedings of Euro PVM/MPI Conference, 2003.
3. M. Krishna, A. Kumar, N. Jayam, G. Senthilkumar, P.K. Baruah, S. Kapoor, R. Sharma, and A. Srinivasan, *A Buffered Mode MPI Implementation for the Cell BE Processor*, to appear in Proceedings of the International Conference on Computational Science (ICCS), Lecture Notes in Computer Science (2007).
4. J. Liu, A.R. Mamidala, and D.K. Panda, *Fast and Scalable MPI-Level Broadcast Using Infiniband's Hardware Multicast Support*, Proceedings of IPDPS, 2004.
5. A.R. Mamidala, L. Chai, H-W. Jin, and D.K. Panda, *Efficient SMP-Aware MPI-Level Broadcast over Infiniband's Hardware Multicast*, Communication Architecture for Clusters Workshop, in Proceedings of IPDPS, 2006
6. A. Mamidala, J. Liu, and D. K. Panda, *Efficient Barrier and Allreduce on IBA Clusters Using Hardware Multicast and Adaptive Algorithms*, Proceedings of IEEE Cluster Computing, 2004.
7. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, *MPI Microtask for Programming the Cell Broadband Engine^{TM} Processor*, IBM Systems Journal, 45 (2006) 85-102.
8. S. Saini, at. al., *Performance Comparison of Cray X1 and Cray Opteron Cluster With Other Leading Platforms Using HPCC and IMB Benchmarks*, NAS Technical Report NAS-06-017, 2006.
9. S. Sistare, R. vande Vaart, and E. Loh, *Optimization of MPI Collectives on Clusters of Large-Scale SMP's*, Proceedings of SC'99, 1999.
10. R. Thakur, R. Rabenseifner, and W. Gropp, *Optimization of Collective Communication Operations in MPICH*, International Journal of High Performance Computing Applications, 19 (2005) 49-66.
11. V. Tipparaju, J. Nieplocha, and D.K. Panda, *Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters*, Proceedings of IPDPS, 2003.
12. W. Yu, D. Buntinas, R.L. Graham, and D.K. Panda, *Efficient and Scalable Barrier over Quadrics and Myrinet with a New NIC-Based Collective Message Passing Protocol*, Workshop on Communication Architecture for Clusters, in Proceedings of IPDPS, 2004.