A Synchronous Mode MPI Implementation on the Cell BE™ Architecture

Murali Krishna[1], Arun Kumar[1], Naresh Jayam[1], Ganapathy Senthilkumar[1],
Pallav K Baruah[1], Raghunath Sharma[1], Shakti Kapoor[2], Ashok Srinivasan[3]

[1]Dept. of Mathematics and Computer Science, Sri Sathya Sai Institute of Higher Learning, Prashanthi Nilayam, India.

[2]IBM, Austin.

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

Abstract

The Cell Broadband Engine is a new heterogeneous multi-core processor from IBM, Sony, and Toshiba. It contains eight co-processors, called SPEs, which operate directly on distinct 256 KB local stores, and also have access to a shared 512 MB to 2 GB main memory. The combined peak speed of the SPEs is 204.8 Gflop/s in single precision and 14.64 Gflop/s in double precision. There is, therefore, much interest in using the Cell for high performance computing applications. However, the unconventional architecture of the SPEs, in particular their local store, create some programming challenges. We describe our implementation of certain core features of MPI, which can help meet these challenges, by enabling a large class of MPI applications to be ported to the Cell. This implementation views each SPE as a node for an MPI process, with the local store abstracted in the library, and thus hidden from the application with respect to MPI calls. We further present experimental results on the Cell hardware, where it demonstrates good performance, such as throughput up to 6.01 GB/s and latency as low as 0.65 μs on the pingpong test. The significance of this work lies in (i) demonstrating that the Cell has good potential for running intra-Cell MPI applications, and (ii) enabling such applications to be ported to the Cell with minimal effort.

Keywords: Cell processor, heterogeneous multi-core processors, MPI.

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). Each SPE has a 256 KB memory called its local store, and access to a shared 512 MB to 2 GB main memory. 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.

Even though the Cell was aimed at the Sony PlayStation3, there has been much interest in using it for High Performance Computing, due to the high flop rates it provides. Preliminary studies have demonstrated its effectiveness for important computation kernels [17]. However, a major drawback of the Cell is its unconventional programming model; applications do need significant changes to fully exploit the novel architecture. Since there exists a large code base of MPI

applications, and much programming expertise in MPI in the High Performance Computing community, our solution to the programming problem is to provide an MPI 1 implementation that uses each SPE as if it were a node for an MPI process.

In implementing MPI, it is tempting to view the main memory as the shared memory of an SMP, and hide the local store from the application. This will alleviate the challenges of programming the Cell. However, there are several challenges to overcome. Some of these require new features to the compiler and the Linux implementation on the Cell. We have addressed the others in this paper, related to the MPI implementation.

In order for an MPI application to be ported to the Cell processor, we need to deal with the small local stores on the SPEs. If the application data is very large, then the local store needs to be used as software-controlled cache and data-on-demand, with the actual data in main memory. We expect some of these features in the next releases of the compiler and operating system [1,2]. These will allow applications to be ported in a generic manner, without hand-coded changes. Meanwhile, in order to evaluate the performance of our implementation, we have hand-coded these transformations for two applications with large data. We have also developed a version of our MPI implementation for small memory applications, which can be ported without being affected by the smallness of the local store.

Empirical results show that our implementation achieves good performance, with throughput as high as *6.01* GBytes/s and latency as low as *0.65* μs on the pingpong test. We expect the impact of this work to be broader than just for the Cell processor due to the promise of heterogeneous multicore processors in the future, which will likely consist of large numbers of simple cores as on the Cell.

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 our implementation in §3 and prove its correctness in §4. This is followed by a presentation of the performance evaluation results in §5. We discuss related work, on Cell applications and on shared-memory MPI implementations, in §6. We then describe limitations of the current work, and future work to overcome these limitations, in §7, followed by conclusions in §8.

2. Cell Architecture

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 each ring, including more than 100 outstanding DMA memory requests between main storage and the SPEs. Simultaneous transfers on the same ring are also possible. The EIB's maximum intra-chip bandwidth is 204.8 GB/s.

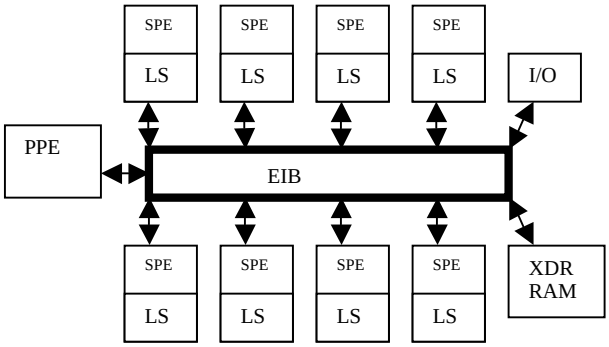
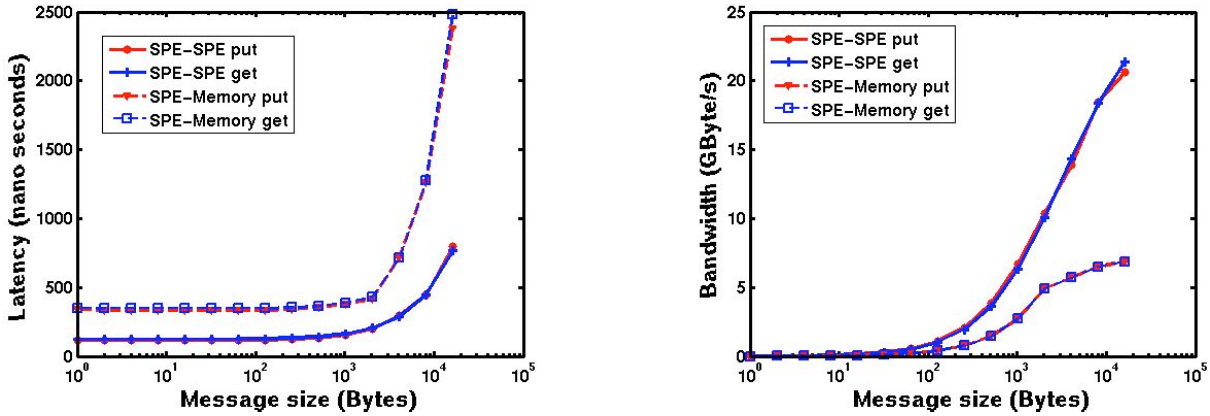


Figure 1: *Overview of cell architecture.*

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 [6] (page 75, table 3.3). *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.* DMA commands may be executed out-of-order.

In order to use the SPEs, a process running on the PPE can spawn a thread that runs on the SPE. 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 the memory mapped address of the destination SPE, and performing a DMA.

We present some performance results in fig. 2 for DMA times. We can see that the SPE-SPE DMAs are much faster than SPE-main memory DMAs. The scale of this difference is much higher than expected, and we need to investigate this further. DMAs to certain special registers can be performed faster than the DMAs presented in fig. 2. For example, an SPE can write to the signal register of another SPE in around *50* ns. We use this feature in our MPI implementation.



(a) (b)

Figure 2: *Latency and bandwidth of DMA operations*

3. MPI Design

In this section, we describe our basic design for blocking point to point communication. We implemented collective communication on top of the point to point communication. We also describe the application start-up process. We have not described the handling of errors, in order to present a clearer high-level view of our implementation.

3.1 MPI Communication Modes

MPI provides different options for the communication mode chosen for the basic blocking point to point operations, MPI_Send and MPI_Recv. We briefly explain our choice below. Implementations typically use either the buffered mode or the synchronous mode. A safe application should not make any assumption on the choice made by the implementation[1] [15].

In the buffered mode, the message to be sent is copied into a buffer, and then the call can return. Thus, the send operation is local, and can complete before the matching receive operation has been posted. Implementations often use this for small messages [3]. For large messages, they avoid the extra buffer copy overhead by using synchronous mode. Here, the send can complete only after the matching receive has been posted. The rendezvous protocol is typically used, where the receive copies from the send buffer to the receive buffer without an intermediate buffer, and then both operations complete. We use the synchronous mode for both short and long messages[2].

3.2 MPI Initialization

A user can run an MPI application, provided it uses only features that we have currently implemented, by compiling the application for the SPE and executing the following command on the PPE:

mpirun –n <N> executable arguments

where <*N*> is the number of SPEs on which the code is to be run. The mpirun process spawns the desired number of threads on the SPE. Note that *only one thread can be spawned on an SPE*, and so <*N*> cannot exceed eight on a single processor or sixteen for a blade. We have not considered latencies related to the NUMA aspects of the architecture in the latter case.

Note that the data for each SPE thread is distinct, and not shared, unlike in conventional threads. The MPI operations need some common shared space through which they can communicate, as explained later. This space is allocated by mpirun. This information, along with other information, such as the rank in MPI_COMM_WORLD, the effective address of the signal registers on each SPE, and the command line arguments, are passed to the SPE threads by storing them in a structure and sending a mailbox message[3] with the address of this structure. The SPE threads receive this information during their call to MPI_Init. The PPE process is not further involved in the application until the threads terminate, when it cleans up allocated memory and then terminates. It is important to keep the PPE as free as possible for good performance, because it can otherwise become a bottleneck.

---

[1] Other specific send calls are available for applications that desire a particular semantic.

[2] We have also implemented an MPI version that uses buffered mode, and verified its correctness and performance. However, we have not optimized its performance, and so do not describe it here.

[3] A mailbox message is a fast communication mechanism for small messages of 32 bits.

3.3 Point-to-point communication

Communication architecture: Associated with each message is meta-data that contains the following information about the message: Address of the memory location that contains the data[4], sender's rank, tag, message size, datatype ID, MPI communicator ID, and an error field. For each pair of SPE threads, we allocate space for two meta-data entries, one in each of the SPE local stores, for a total of *N(N-1)* entries, with (N-1) entries in each SPE local store; entry B_{ij} is used to store meta-data for a message from process *i* to process *j*, $i \neq j$. Such schemes are used by other implementations too, and it is observed that it is not scalable for large *N*. However, here *N* is limited to eight for one processor, and sixteen for a blade (consisting of two processors). Each meta-data entry is *128* bytes, and so the memory used is small. It has the advantage that it is fast.

Send protocol: The send operation from P_i to P_j proceeds as follows. The send operation puts the meta-data entry into buffer B_{ij} through a DMA operation.

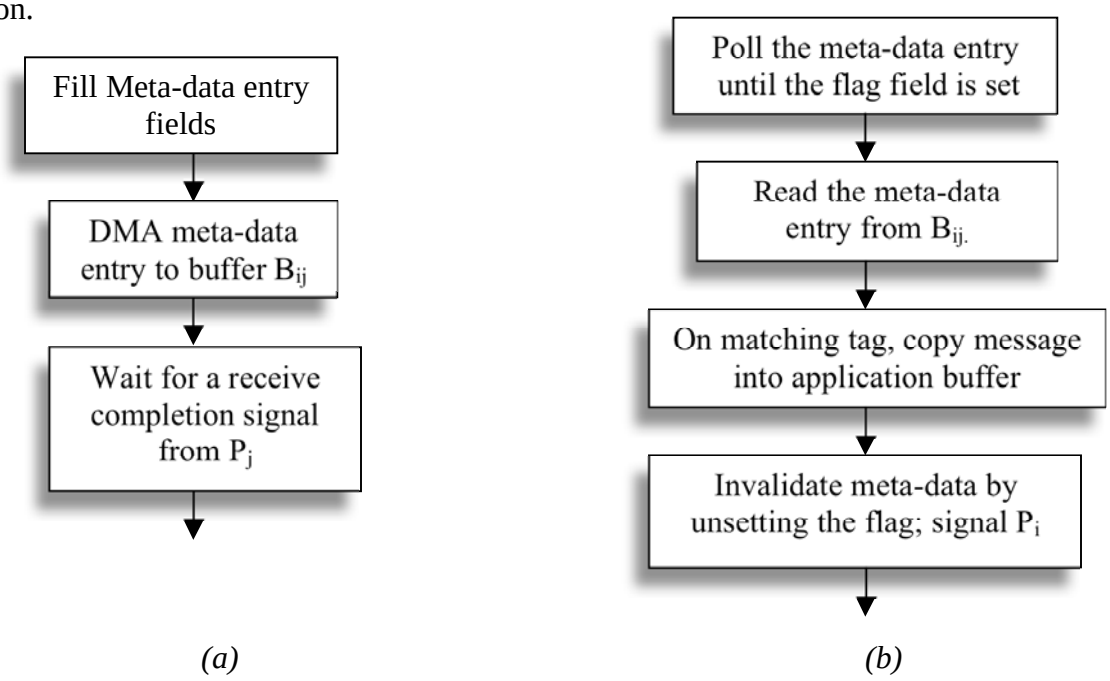


Figure 3: *Execution of (a) send and (b) receive operations from a specific source SPE P_i to P_j.*

The send operation then waits for a signal from P_j notifying that P_j has copied the message. The signal obtained from P_j contains the error value. It is set to MPI_SUCCESS on successful completion of the receive operation and the appropriate error value on failure. In the synchronous mode, an SPE is waiting for acknowledgment for exactly one send, at any given point in time, and so all the bits of the receive signal register can be used. Figure 3 shows the flow of the send operation.

Receive protocol: The receive operation has four flavors. (i) It can receive a message with a specific tag from a specific source, (ii) it can receive a message with any tag (MPI_ANY_TAG) from a specific source, (iii) it can receive a message with a specific tag from any source (MPI_ANY_SOURCE), or (iv) it can receive a message with any tag from any source.

[4] The address is an effective address in main memory. In the modification made for small memory applications, it is the effective address of a location in local store.

The first case above is the most common, and is illustrated in fig. 3. First, the meta-data entry in B_{ij} is continuously polled, until the flag field is set. The tag value in the meta-data entry is checked. If the application truly did not assume any particular communication mode, then this tag should match, and the check is superfluous. However, correct but unsafe applications may assume buffered mode for short messages, and so we perform this check[5]. If the tag matched, then the address of the message is obtained form the meta-data entry.

Receive then transfers data from the source SPE's application buffer to its buffer and signals P_i's signal register to indicate that the data has been copied. Note that the data transfer is a little complicated, because the SPE cannot perform a *memcpy*, since source and destination buffers are in main memory, and not in its local store. It also cannot DMA from main memory to main memory, because it can only DMA between local store and some effective address. So we DMA the data to local store and then transfer it back to its location in main memory. For large data, this is performed in chunks, with double buffering being used to reduce the latency to a certain extent[6]. After the receiving SPE completes the transfer, it unsets the meta-data flag field, and then signals the sending SPE's signal register.

While the data transfer process described above appears wasteful, a similar situation occurs in traditional cache-based processors too [12]; *memcpy* brings the source and destination to cache from main memory, copies the data, and writes the destination back to cache, incurring three cache misses. Some systems have alternate commands to avoid at least one of these misses. But these cache misses have a dominant effect on MPI performance on shared memory systems.

The second case is handled in a manner similar to the first, except that any tag matches.

The third and fourth cases are similar to the first two respectively, as far as tags are concerned. However, messages from any source can match. So the receive operation checks the meta-data entry flags for each sender, repeatedly, in a round robin fashion, to avoid starvation, even though the MPI standard does not guarantee against starvation. If any of the flags is set, and the tag matches for case (iii), then the rest of the receive operation is performed as above. Note that in case (iii), it is not an error for a tag not to match, because there may be a message from another sender that it should match.

3.4 Collective communication

Our collective communication calls are built on the send and receive calls. The focus has been on providing functionality rather than to optimize it for performance at this time. For example, in the broadcast operation, the root performs N-1 send operations, while the other nodes perform a receive operation. Other collective communication calls are also implemented in a somewhat crude manner.

4. Proof of Correctness

In this section, we prove the correctness of our design, and verify the correctness of our implementation.

We verified the correctness of our implementation by running tests provided by MPICH. These tests not only check for correct implementation when the messages are correct, but also check for

[5] We do not mention other error checks that are fairly obvious.

[6] For small memory applications, both buffers are in local stores. Since a local store address can be mapped to an effective address, a single DMA suffices.

correct error handling, as required by the specifications. We performed tests relevant to the features that we have implemented, considering the synchronous mode that we are using. In particular, we excluded a test that checks to see if a message with a matching tag is allowed to overtake an earlier message with a non-matching tag, because this is not permitted in the synchronous mode. We tested our version for the small-application model, because the general model for large memory would require changing the code for each test. Since the implementations for both the models are identical, except for a minor detail on the manner in which data transfer occurs, we can expect the general model too to be correct. We have also verified the correctness of both models on the applications considered, and on other tests where we tested the individual routines.

We next show the correctness of the design by proving some properties of the messages. Before we do that, we summarize some requirements of messages in MPI [15].

Requirements: (i) *Message ordering:* messages from the same processor should be non-overtaking, in the sense that if two messages sent by the same processor can match a receive, then the later message should not be received if the first one has not been received. A similar requirement is made on the receives. However, causal ordering of messages from different processors is not required. For example, if processor 1 sends message A to processor 3 and then message B to processor 2, and if processor 2 sends message C to processor 3 after receiving message B, then it is acceptable for processor 3 to receive message C before message A, even though message C occurred causally after message A. (ii) *Progress:* If process P_i sends a message A to P_j, and P_j posts a receive that can match A, then at least one of the two operations, the send or the receive, should complete. (The send may not complete if the receive matched a previous send and vice versa.)

Assumptions: We assume that the underlying communication system is reliable. We assume that the application program itself is *correct and safe* according to MPI specifications. In particular, it should satisfy the following two requirements. (i) The application protocol itself should be free of deadlock and starvation. (ii) The application should not assume any particular communication mode. Consequently, when MPI_ANY_SOURCE is not used, a send call from P_i to P_j should be matched by a corresponding receive in the same order. MPI_ANY_SOURCE can make the message matching non-deterministic, but that is acceptable, as long as the application did not assume a buffered communication mode for correctness.

Theorem 4.1: (Meta-Data Properties) A meta-data entry has the following properties.

a) Meta-data of a later message cannot overtake the meta-data of an earlier message, or over-write it before the receive for the first message completes.

b) The same meta-data entry will not be read by more than one receive call.

Proof: Consider two messages M_1 and M_2 from sender P_i to receiver P_j where M_1 is sent first and M_2 is sent latter. According to the protocol described in §3.3 the send operation for M_1 waits for a signal from the receiver before returning control to the application. Also the receive operation signals the sender only after reading the meta-data entry and copying the message into the application buffer. The send operation for M_2 will be posted only after the send operation for M_1. Therefore meta-data entry M_2 will not be written until the meta-data entry for M_1 is read by the receive operation. So M_1's meta-data is not over-written by M_2 until its receive has

completed. Consequently, messages are not lost. Since M_2 does not start until M_1 has completed, its metadata cannot overtake that of M_1. Hence claim (a) is proved.

Now, to prove the second property, consider the receive operation. As described in §3.3, the receive operation unsets the flag field in the meta-data entry before signaling the sender about the completion of receive operation. Therefore, if a receive operation is posted again for a message of the same tag, the meta-data entry will not be read until the flag bit is set by the sender. Also, from the send protocol, this will be set as a consequence of the sender writing a new meta-data entry in the buffer. Hence, two receives will not read the same meta-data entry, proving claim (b).

Theorem 4.2: (Message Ordering for sends) If two messages sent by the same processor can match a receive, then the later message should not be received if the first one has not been received.

Proof: Consider the case where a sender P_i sends two messages M_1 and then M_2 that can match a receive R_1 on P_j. We claim that R_1 cannot receive M_2 unless M_1 has already been received. From theorem 4.1 (a), the meta-data of M_2 cannot overtake that of M_1 or over-write it before M_1 is received. Also, a message cannot be matched until after its meta-data is entered in the meta-data buffer. From the above two properties, R_1 cannot receive M_2 unless M_1 has already been received.

Theorem 4.3: (Message Ordering for receives) If two receives on the same processor can match a sent message, then the later receive will not match the message unless the first receive has already completed.

Proof: This follows immediately from the synchronous mode, since the second receive cannot start until the first one has been satisfied.

Theorem 4.3: (Progress). If process P_i sends a message M to P_j, and P_j posts a receive R that can match M, then at least one of M or R completes.

Proof: We prove this by contradiction. Assume that neither M nor R complete. Then M is either (i) waiting for its DMA of the meta-data to B_{ij} to complete or (ii) waiting for the acknowledgement signal from P_j, and R is waiting for either (i) the flag field of the meta-data entry to be set or (ii) waiting for the copying of the send buffer to the destination buffer to complete. Note that the DMA to B_{ij} will eventually complete. So M will eventually stop waiting for its condition (i), and will then send a signal to P_j. So R will eventually stop waiting for its condition (i), and will then start a copy operation (which may not necessarily be to the message from P_i, if MPI_ANY_SOURCE was used). The copy operation performs DMA operations that will eventually complete, after which R will stop waiting for its condition (ii). R will then send a signal to the sending process and then complete. So our assumption is false, and the theorem is proven.

5. Performance Evaluation

The purpose of the performance evaluation is to first show that our MPI implementation achieves performance comparable to good MPI implementations, even though the SPEs are not full-fledged cores. In particular, we will compare with shared memory intra-node MPI implementations, which have low latencies and high throughputs. We next demonstrate that it scales well in large memory applications, where the application data is in main memory.

We performed our experiments on a 3.2 GHz Rev 2 Cell blade with 1 GB main memory running Linux 2.6.16 at IBM Austin. We had dedicated access to the machine while running our tests.

5.1 Communication Performance

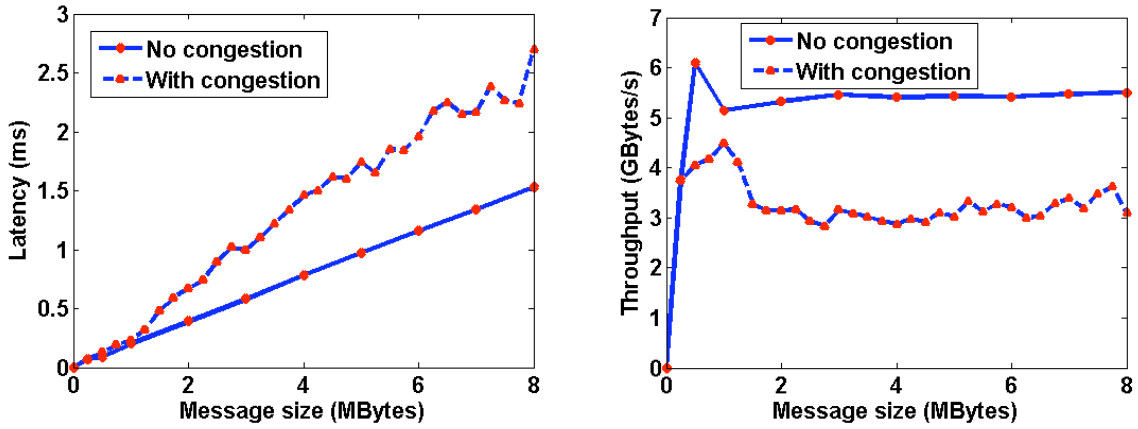
Figure 5 shows the latency and bandwidth results using the pingpong test from mpptest [11]. It was modified to place its data in main memory, instead of in local store. We determined the wall clock time for these operations by using the decrementer register in hardware, which is decremented at a frequency of 14.318 MHz, or, equivalently, around every 70 ns. Between calls to the timer, we repeated each pingpong test loop 1000 times. For the shortest latencies we obtained (0.65 μs per message), this yields an accuracy of around *0.005%* (observing that each iteration of the loop involves two messages). The accuracy is greater for the larger tests.

We performed each test multiple times and took the average. Note that it is sometimes recommended that one take the smallest of the times from multiple experiments to obtain reproducible results [11]. This would give number a little smaller than the average that we report. However, our results were quite close to each other, and so there is not much of a difference between the average and the minimum.

Note that on cache-based processors, one needs to ensure that the tests do not already have their data in cache. However, the SPEs do not have a cache, and so this is not an issue. The implementation transfers data between main memory locations, using the SPE local store as a temporary location (analogous to a cache miss). The small-application tests move data from local store to local store.

Figure 5 (a) shows the latency results for point to point communication on the pingpong test, in the presence and absence of congestion. The congested test involved dividing the SPEs into pairs, and having each pair exchanging messages. One pair was timed. In the non-congested case, only one pair of SPEs communicated. The smallest latency for the non-congested case is around 0.65 μs. The 0-byte message incurs the following costs: one SPE-SPE DMA, for the meta-data on the sender side, and one SPE_SPE signal on the receive side (the other DMAs, for getting data from main-memory to local store on the receiving side, and once again for sending that data back from local store to main memory, do not occur with a 0-byte message). Each SPE-SPE DMA takes a little under 0.12 μs and the SPE-SPE signal takes around 0.05 μs.

Figure 5 (b) shows the throughput results for the same tests as above for large messages, where the overhead of exchanging meta-data and signals can be expected to be relatively insignificant. The maximum throughput observed is around 6 GB/s. From fig. 2, we can observe that the maximum bandwidth for blocking SPE to main memory DMAs is around 6 GB/s. Note that though we need to transfer each data twice, once from main memory to SPE and then from SPE to main memory, due to double buffering and the ability to have multiple DMAs simultaneously in transit, the effective time is just half of the round-trip time. This is possible because at 6 GB/s, we have not saturated the bandwidth of 25.6 GB/s available to an SPE. Thus the observed DMA time agrees with what one would expect from the DMA times. However, the DMA time is well below the 25.6 GB/s that is expected and we need to study this further. Fixing this problem, for example, by having more DMAs simultaneously in flight, may enable our implementation to achieve an improvement by a factor of two over the times reported here.



(a) (b)

Figure 5: *Latency and throughput results for point-to-point communication.*

MPI/Platform	Latency (0 Byte)	Maximum throughput
Cell	0.65 μs	6.01 GB/s
Cell Congested	NA	4.48 GB/s
Cell Small	0.65 μs	23.29 GB/s
Nemesis/Xeon	≈ 1.0 μs	≈ 0.65 GB/s
Shm/Xeon	≈ 1.3 μs	≈ 0.5 GB/s
Open MPI/Xeon	≈ 2.1 μs	≈ 0.5 GB/s
Nemesis/Opteron	≈ 0.34 μs	≈ 1.5 GB/s
Open MPI/Opteron	≈ 0.6 μs	≈ 1.0 GB/s
TMPI/Origin 2000	≈ 15.0 μs	≈ 0.115 GB/s

The table on the left compares the latencies and bandwidths with those of some good intra-node implementations on other hardware. We can see that MPI on the Cell has performance comparable to those on processors with full-fledged cores.

In table 1, *Cell* refers to our implementation on the Cell processor. *Cell small* refers to our implementation for small applications. *Nemesis* refers to the MPICH using the Nemesis communication subsystem. *Open MPI* refers to the Open MPI implementation. *Shm* refers MPICH using the shared-memory (shm) channel. *TMPI* refers to the Threaded MPI implementation on an SGI Origin 2000 system reported in [16]. *Xeon* refer to timings on a dual-SMP 2 GHz Xeon, reported in [5]. *Opteron* refers to timings on a 2 GHz dual-core Opteron, reported in [3].

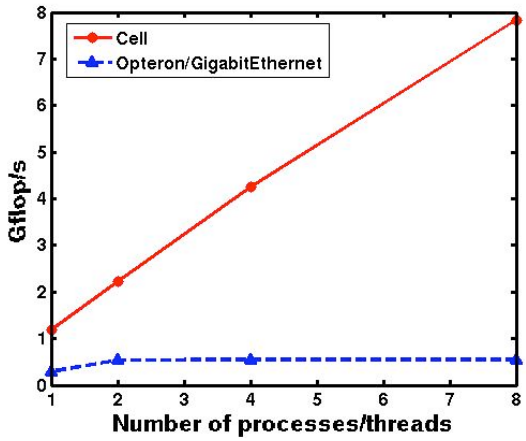
5.2 Application Performance

We demonstrated the use of our MPI implementation on large memory applications by transforming two applications, double precision matrix multiplication and matrix-vector multiplication. Space in main memory is allocated by the PPE during initialization, in a fairly automatic manner. But the application code had to be changed while accessing these data, to perform DMA reads and writes.

The matrix multiplication algorithm that we used was not an efficient one, but rather, was the algorithm from [10], which is meant to demonstrate MPI programming techniques. Transforming an optimized algorithm would have required much more effort. We also did not modify the application to use SPE specific intrinsics, which would have given us much better performance. This algorithm works in a master-worker paradigm. In order to determine C = A × B, each

worker SPE stores an entire copy of B. The master repeatedly sends a row to each processor, and then receives the result from each processor. We could achieve a double precision throughput of 5.85 Gflop/s for matrices of size 512 and 5.66 Gflop/s for matrices of size 1024. While this is low compared to the peak performance of 14.64 Gflop/s, the Cell still gives better results than on an Opteron processor, for the same algorithm. A 2 GHz dual processor (not dual core) Opteron with Gigabit interconnect yielded 40 Mflop/s for the same algorithm with one worker processor[7], and 210Mflop/s with seven worker processors (the same as for the Cell).

The MPI algorithm on the Cell runs twenty five to hundred times faster than on the Opteron cluster, depending on which Opteron data we use to compare. We could compare flops per dollar, flops per process/thread, flops per watt, flops per transistor, or flops per chip. If we compared per process/thread, then the Cell would be around twenty five times faster. If we compared per transistor or per chip, the Cell would be around a hundred times faster.



The matrix vector multiplication algorithm to compute c = A × b is a standard parallelization with a 1-D decomposition, where c and b are replicated on all processors, and A is distributed row-wise. An AllGather is required to replicate c, and is the only communication. We did not use SPE intrinsics to optimize this application either, since our focus is on the parallelization. We can see from fig. 6 that the implementation on the Cell outperforms that on the Opteron cluster significantly. Again, the difference is one or two orders of magnitude, depending on the criterion used.

Figure 6: *Performance on matrix-vector multiplication.*

6. Related work

6.1 Shared Memory MPI Implementations

Conventional shared memory MPI implementations run a separate process on each processor. These processes have distinct address spaces. However, operating systems provide some mechanisms for processes to be allocated shared memory regions, which can be used for fast communication between processes. There are a variety of techniques that can be used, based on this general idea. They differ in scalability, effects on cache, and latency overhead. A detailed comparison of popular techniques is presented in [4]. Among the current implementations, the Nemesis communication subsystem [3,5] appears to give some of the best results. It consists of

[7] Incidentally, an optimized BLAS implementation on a single 3.2GHz 3GB Xeon processor at NCSA yields 5 Gflop (www.ncsa.uiuc.edu/UserInfo/Perf/BLAS/tungsten.html) on matrix multiplication, and 3 Gflop/s on matrix-vector multiplication. An optimized double precision matrix multiplication routine on the Cell processor yields over 14 Gflop/s.

*fastbox*es, which are somewhat like buffers B_{ij} in our implementation, except that when a fastbox is occupied, scalable (O(N)) lock-free queues are used, thus permitting the buffered mode.

The TMPI implementation takes a slightly different approach, by spawning threads instead of processes [16]. Since the global variables of these threads are shared (unlike that of the SPE threads in our implementation), some program transformation is required to deal with these. They too use a lock-free queue, but they use $O(N^2)$ queues.

Note that some implementations on conventional processors need memory barriers to deal with out of order execution, which is common on modern processors [9]. In order execution on the Cell avoids such problems.

6.2 Cell Applications and Programming

There has been much interest in exploring the potential of the Cell processor for High Performance Computing applications, and on developing suitable programming paradigms for easing the programming burden.

Work has been done to port a number of computational kernels like DGEMM, SGEMM, 1D FFT and 2D FFT to the cell processor [17], and close to peak performance is obtained on DGEMM. Results on the other kernels too show results much superior to those on conventional processors.

Much work is being performed to make the programming of the Cell processor easier, such as developing frameworks that will enable the programming of the Cell at an abstract level [14,8,13]. Since the local store of each of the SPE is too small for most of the applications, research on developing compiler and runtime support for running programs with data and text larger than the size of the local store is being carried out [1,2]. The use of MPI microtasks was proposed in [14], to make programming simpler. Each microtask would be a small MPI task. This is an attractive option for developing new applications. However, the effort required may still be large to convert an existing application.

7. Limitations and Future Work

We now discuss limitations of the current work, and our future plan to overcome these limitations.

Synchronous mode: While a safe application should not make any assumption regarding the communication mode of the implementation, some applications do assume a buffered mode, especially when dealing with short messages. We have developed another implementation that uses buffered mode for short messages and synchronous mode for long messages.

Users often assume a buffered implementation in a common communication pattern where pairs of processes exchange data by first sending a message and then receiving a message. This can lead to deadlock in the synchronous mode, and is easily avoided by use of the MPI_Sendrecv call. We have implemented a deadlock-free MPI_Sendrecv call on top of send and receive, for users who need it before we make the buffered mode available.

Limited functionality: We have provided some core features of MPI 1, but not the complete set. We plan to implement more features. In order to provide this faster, we will consider implementing a channel or device interface to MPICH. If the code size is too large, then we will provide overlaying facilities in the library itself, if the compiler features are not available by then.

Non-blocking calls: These calls are relatively easy to provide. Both the send and the receive should be made capable of starting the data transfer, with the later call initiating a non-blocking DMA call. The wait calls should wait for the DMA to complete.

Compiler and OS support for the application: As mentioned earlier, the non-MPI portion of the application still needs some compiler and OS support in order to be ported without changes to the code. We expect this to be accomplished by other groups.

8. Conclusions

We have demonstrated that an efficient MPI implementation is possible on the Cell processor, using just the SPEs, even though they are not full-featured cores. Small-memory applications using the core features of MPI can use our implementation right now, without any changes to the code being required. Other applications can either make some hand-coded changes, or wait for compiler and OS support that is expected in the near future. Thus, our approach eases the programming burden, which is considered a significant obstacle to the use of the Cell processor. Furthermore, our implementation demonstrates that simple cores for future generation heterogeneous multicore processors can run MPI applications efficiently.

Acknowledgment

We thank Darius Buntinas of Argonne National Lab, for referring us to appropriate literature. We also thank several employees at IBM Bangalore for running the performance tests on the Cell hardware. Most of all, we express our gratitude to Bhagavan Sri Sathya Sai Baba, Chancellor of Sri Sathya Sai Institute of Higher Learning for bringing us all together to perform this work, and for inspiring and helping us toward our goal.

References

[1] *An Introduction to Compiling for the Cell Broadband Engine Architecture, Part 4: Partitioning Large Tasks,* Feb 2006. http://www-128.ibm.com/developerworks/edu/pa-dw-pa-cbecompile4-i.html

[2] *An Introduction to Compiling for the Cell Broadband Engine Architecture, Part 5: Managing Memory,* Feb 2006. Analyzing calling frequencies for maximum SPE partitioning optimization. http://www-128.ibm.com/developerworks/edu/pa-dw-pa-cbecompile5-i.html

[3] D. Buntinas, G. Mercier and W. Gropp, *Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem*, Proceedings of the Euro PVM/MPI Conference, 2006.

[4] D. Buntinas, G. Mercier and W. Gropp, *Data Transfers Between Processes in an SMP System: Performance Study and Application to MPI*, Proceedings of the International Conference on Parallel Processing, (2006) 487-496.

[5] D. Buntinas, G. Mercier and W. Gropp, *Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem*, Proceedings of the International Symposium on Cluster Computing and the Grid, 2006.

[6] *Cell Broadband Engine Programming Handbook,* Version 1.0 April 19, 2006. http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/9F820A5FFA3ECE8C8725716A0062585F/$file/BE_Handbook_v1.0_10May2006.pdf

[7] T. Chen, R. Raghavan, J. Dale, and E. Iwata. *Cell Broadband Engine Architecture and its First Implementation.* November 2005. http://www.ibm.com/developerworks/ power/library/pa-cellperf/

[8] K. Fatahalian, T. J. Knight, M. Houston, and M. Erez, *Sequoia: Programming the Memory Hierarchy*, Proceedings of SC2006.

[9] W. Gropp and E. Lusk, *A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer*, Parallel Computing, 22 (1997) 1513-1526.

[10] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1999.

[11] W. Gropp and E. Lusk, *Reproducible Measurements of MPI Performance Characteristics*, Argonne National Lab Technical Report ANL/MCS/CP-99345, 1999.

[12] H.-W. Jin and D. K. Panda, *LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster*, Proceedings of the International Conference on Parallel Processing, (2005) 184-191.

[13] MultiCore Framework, *Harnessing the Performance of the Cell BE™ Processor*, Mercury Computer Systems, Inc., 2006. http://www.mc.com/literature/literature_files/MCF-ds.pdf

[14] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, *MPI Microtask for Programming the Cell Broadband Engine™ Processor*, IBM Systems Journal, 45 (2006) 85-102.

[15] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI – The Complete Reference*, Volume 1, The MPI Core, second edition, MIT Press 1998.

[16] H. Tang, K. Shen, and T. Yang, *Program Transformation and Runtime Support for Threaded MPI Execution on Shared-Memory Machines*, ACM Transactions on Programming Languages and Systems, 22 (2000) 673-700.

[17] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, The *Potential of the Cell Processor for Scientific Computing*, Proceedings of the ACM International Conference on Computing Frontiers, 2006.