A Buffered-Mode MPI Implementation for the Cell BE™ Processor

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

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

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

[3]IBM, Austin.

Abstract. The Cell Broadband Engine™ is a heterogeneous multi-core architecture developed by IBM, Sony and Toshiba. It has eight computation intensive cores (SPEs) with a small local memory, and a single PowerPC core. The SPEs have a total peak single precision performance of 204.8 Gflops/s, and 14.64 Gflops/s in double precision. Therefore, the Cell has a good potential for high performance computing. But the unconventional architecture makes it difficult to program. We propose an implementation of the core features of MPI as a solution to this problem. This can enable a large class of existing applications to be ported to the Cell. Our MPI implementation attains bandwidth up to 6.01 GB/s, and latency as small as 0.41 µs. The significance of our work is in demonstrating the effectiveness of intra-Cell MPI, consequently enabling the porting of MPI applications to the Cell with minimal effort.

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

1 Introduction

The Cell is a heterogeneous multi-core processor targeted at the gaming industry. There is also much interest in using it for high performance computing. Some compute intensive math kernels have shown very good performance on the Cell [1], demonstrating its potential for scientific computing. However, due to its unconventional programming model, applications need to be significantly changed in order to exploit the full potential of the Cell. As a solution to the programming problem, we provide an implementation of core features of MPI 1, which uses each SPE as if it were a node for an MPI process. This will enable the running of the large code base of existing MPI applications.

Each SPE has a small (256 KB) local store that it can directly operate on, and access to a larger common main memory from/to which it can DMA data. If one attempts to directly port an application to the SPE, then the small size of the SPE local store poses a significant problem. For applications with data larger than the local store size, a software-controlled cache approach is feasible, wherein the data is actually in the main memory and moved to the local store as needed. Some features to automate this process are expected in the next releases of the compiler and operating system [2,3]. These features would allow the porting of applications in a more generic fashion, except for the parallel use of all the SPEs. Our MPI implementation handles the parallelization aspect too. We have hand-coded some large-data applications in order to analyze the performance of our implementation. For small memory applications, we have modified our MPI implementation so that such applications can be ported right now, without needing the compiler and operating system support expected in the near future.

Existing MPI implementations for the shared memory architectures cannot be directly ported to the Cell, because the SPEs have somewhat limited functionality. For example, they cannot directly operate on the main memory – they need to explicitly DMA the required data to local store and then use it. In fact, they cannot even dynamically allocate space in main memory.

Heterogeneous multi-core processors show much promise in the future. In that perspective, we expect the impact of this work to be much broader than just for the Cell processor, by demonstrating the feasibility of MPI on heterogeneous multi-core processors.

The rest of the paper is organized as follows. In §2, the architectural features of the Cell processor that are relevant to the MPI implementation are described. Our MPI implementation is described in §3. The performance results are presented in §4. We discuss related work in §5. We then describe limitations of the current work, and future plans to overcome these limitations, in §6, followed by conclusions in §7.

2 Cell Architecture

The Cell processor consists of a cache coherent PowerPC core and eight SPEs running at 3.2 GHz. All of them 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, SPEs, DRAM controller, and I/O controllers are all connected via four data rings, collectively known as the EIB. Multiple data transfers can be in progress 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.

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 [4] (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 space of the process that spawned the SPE thread. Data can be transferred from the local store of one SPE to the local store or special registers of another SPE by obtaining these memory mapped addresses.

3 MPI Design

In this section, we describe our basic design for the blocking 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 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 SPEs. 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, and the command line arguments, are passed to the SPE threads by storing them in a structure and sending a mailbox message[1] 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.

3.2 Point-to-point communication

We describe the implementation of point-to-point communication. Collective operations were implemented on top of the point-to-point operations.

Communication Architecture. Let *N* be the number of MPI nodes. The mpirun process allocates *N* message buffers in main memory in its address space. Since the SPE threads are part of the mpirun process, they can access the buffers allocated by mpirun. Buffer P_i is used by SPE *i* to copy its data, when SPE *i* sends a message. Thus, even though SPEs cannot dynamically allocate space in main memory, they can use space allocated by mpirun. SPE *i* manages the piece of memory P_i, to allocate space within P_i for its send calls.

The sender and receiver processes communicate information about messages through meta-data queues maintained in SPE local stores. There are $N*(N-1)$ queues in total. Queue Q_{ij} is used by SPE *i* to send information about a message to SPE *j*. Q_{ij} is present in the local store of SPE *j* (the receiver). Thus each SPE maintains *N-1* queues. These queues are organized as circular arrays. Each entry in Q_{ij} contains information about the location of the message within P_i, the message tag, the data type, message size, communicator identifier and flag bits. The total size of an entry is

[1] A mailbox message is a fast communication mechanism for 32-bit messages.

16 bytes. The flag field is either *valid*, indicating that the entry contains information on a sent message that has not been received, or is *invalid*, indicating that it is free to be written by the sending SPE. All entries are initialized to *invalid*.

Send Protocol. The send operation from SPE *i* to SPE *j*, shown in fig. 1 (a), proceeds as follows: The send operation first finds the offset of a free block in buffer P_i, managed by it. The message data is copied into this location in P_i. The copying is done through DMA operations. Since the SPE's data and P_i are in main memory, the data is first DMA-ed into the local store from main memory in pieces, and then DMA-ed out to P_i. The send operation does a single DMA or a series of DMAs, depending on the size of the message (a single DMA transfer can be of 16KB maximum). SPE *i* then finds the next entry in the meta-data queue Q_{ij} and waits until it is marked *invalid*. This entry is updated by DMA-ing the relevant information, with flag set to *valid*. Send returns after this DMA completes. Note that we use blocking DMAs in copying data, to ensure that P_i contains the entire data before the corresponding entry has flag set to *valid*.

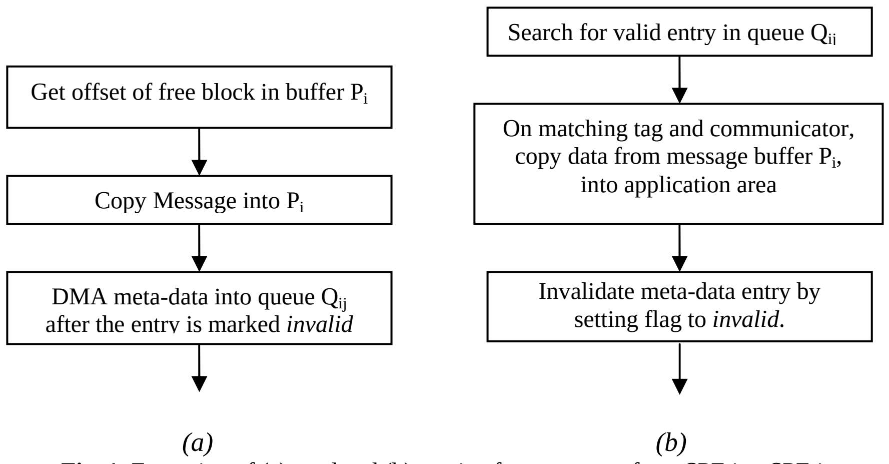
Get offset of free block in buffer P_i
Copy Message into P_i
DMA meta-data into queue Q_{ij} after the entry is marked *invalid*
Search for valid entry in queue Q_{ij}
On matching tag and communicator, copy data from message buffer P_i, into application area
Invalidate meta-data entry by setting flag to *invalid*.
(a)
(b)

Fig. 1. Execution of (a) send and (b) receive for a message from SPE i to SPE j.

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. Case (i) is shown in fig. 1 (b).

The receive operation on SPE *j* for a message from SPE *i* proceeds as follows, in case (i). The meta-data queue Q_{ij}, is searched in order to find a valid entry with the specific tag and communicator value. The searching is done from the logical front of the circular array to the logical end, in a linear order, to avoid overtaking of an older entry by a newer one with the same tag value. The search is repeated until a matching entry is found. Once a matching entry is found, the location of the message in main memory is obtained from the location field of the meta-data entry and the data is copied from P_i into the location for the application variable, in a similar manner as in the send operation. Finally, the meta-data entry is marked invalid.

In case of MPI_ANY_TAG by a receiver *j* from a specific source *i*, the first valid entry in Q_{ij} with the same communicator is matched. MPI_ANY_SOURCE has two cases similar to the above, except that queues Q_{ij}, for each *i*, are searched.

Lock Free Data Structure. The meta-data queues are handled in a lock free approach. Each queue is an array of entries which is filled by the sender in a circular fashion. The receiver maintains the range of the entries to be searched, in its local store. The need for locks has been avoided by using the fact that the local store is single ported. That is, at a given clock cycle, either a DMA operation can access the local store or the load store unit of the SPE can access it. DMA writes are in units of 128 bytes. The meta-data entry is less than 128 bytes and will therefore be seen in full, or not seen at all, by the receive operation when the receiver's search for the entry and the sender's DMA of it are taking place simultaneously. (The size of each entry is 16 bytes, which divides 128, and so all entries are properly aligned too.) Therefore the send and receive can operate in a lock free fashion.

Communication modes. MPI_Send may be implemented in either buffered mode, as in the description above, or in synchronous mode. In the latter, the send can complete only after the matching receive has been posted. The rendezvous protocol is typically used, where the receive copies the data directly, without an intermediate buffer, and then both send and receive complete. A safe application should not make any assumption on the choice made by the implementation [5]. Implementations typically use the buffered mode for small messages, and switch to synchronous mode for large messages [6]. We too switch to synchronous mode for large messages. The send then writes the address of the data in main memory into the meta-data entry, and blocks until the receive operation copies this data.

4 Performance Evaluation

We evaluated the performance of our MPI implementation, in order to determine the bandwidth and latency as a function of the message size. We also evaluated the performance of a parallel matrix-vector multiplication kernel using our MPI implementation. 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 Rochester. We had dedicated access to the machine while running our tests.

Figures 2 and 3 show the latency and bandwidth results respectively, using the pingpong test from mpptest [7]. We switch from buffered mode to synchronous mode for messages larger than 2 KB. The pingpong test was modified to place its data in main memory, instead of in local store. We timed the operations by using the decrementer register available in the SPE, which is decremented at a frequency of 14.318 MHz, or, equivalently, around every 70 ns. The latency is comparable to that on good shared memory implementations, such as around 1.1 μs for MPICH with Nemesis on Xeon [8] and around 0.3 μs for the same on an Opteron [6]. The peak bandwidth is 6.01 GB/s, compared with around 0.65 GB/s and around 1.5 GB/s in the latter two respectively. Thus, the performance is comparable to good shared memory implementations on full-fledged cores, even though the SPEs have limited functionality.

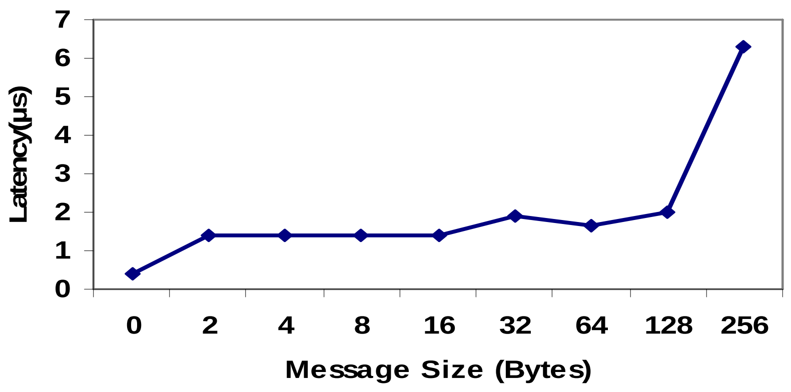

Fig. 2. Latency results.

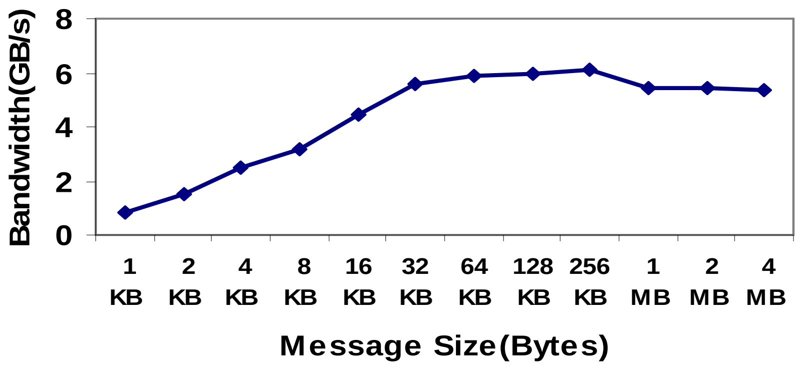

Fig. 3. Bandwidth results.

We also studied the performance of a parallel double precision matrix-vector kernel using a simple 1-dimensional decomposition. We transformed the application by placing the data in the main memory and inserting DMA instructions wherever necessary. We did not use SPE intrinsics to optimize this application, nor was the algorithm chosen optimal, because our focus was on the parallelization. The only MPI communication in this application is an *MPI_Allgather* call. We got a throughput of 6.08 Gflops for a matrix of dimension 512 and throughput of 7.84 Gflops for a square matrix of dimension 1024. The same implementation yields 0.292 Gflops on a single core 2 GHz Opteron, and 0.553 Gflops on 8 Opteron processors/4 nodes connected with Gigabit Ethernet. An optimized BLAS implementation for this kernel on a single 3.2 GHz 3GB RAM Xeon processor at NCSA yields 3 Gflops.

5 Related Work

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 [9]. The TMPI implementation takes a slightly different approach, by spawning threads instead of processes [10]. 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 $O(N^2)$ lock-free queues, but the implementation differs from ours. Note that some implementations on conventional processors need memory barriers to deal with out of order execution, which is common on modern processors[11]. In-order execution on the Cell avoids such problems.

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 [12,13,14]. Work has also been done to port a number of computational kernels like DGEMM, SGEMM, 1D FFT and 2D FFT to the Cell processor [1], and close to peak performance is obtained on DGEMM. Results on other kernels too show much superior performance to those on conventional processors.

6 Limitations and Future Work

We have implemented some core features of MPI 1, but not the complete set. We plan to implement more features. If the code size is too large, then we intend to provide overlaying facilities in the library itself, which will bring in code to the local store as needed. Also, we intend to implement a customized software cache and study the impact of the latencies introduced due to the use of a software cache and NUMA aspects of the blade. We intend to implement non-blocking calls. Also, we plan to optimize the collective communication calls using Cell-specific features. The non-MPI portion of the application still needs some compiler and OS support in order to be ported without changes to the code for large memory applications. We expect this to be accomplished by other groups.

7 Conclusions

We have shown the feasibility of an efficient MPI implementation on the Cell processor, using the SPEs as MPI nodes. Applications using only the core features of MPI can use our implementation right now, without any changes to the code if the application fits into the local store memory. Large applications can either make some hand-coded changes, or wait for compiler and OS support that is expected in the near future. Our approach, therefore, reduces 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 may run MPI applications efficiently.

Acknowledgments. We thank several employees at IBM Bangalore for running the performance tests on the Cell hardware. Most of all, we express our gratitude to Bhagawan Sri Sathya Sai Baba, Chancellor of Sri Sathya Sai University, for bringing us all together to perform this work, and for inspiring and helping us toward our goal.

References

1. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The Potential of the Cell Processor for Scientific Computing, Proceedings of the ACM International Conference on Computing Frontiers, (2006)
2. An Introduction to Compiling for The Cell Broadband Engine Architecture, Part 4: Partitioning Large Tasks, (2006) http://www-128.ibm.com/developerworks/edu/pa-dw-pa-cbecompile4-i.html
3. An Introduction to Compiling for The Cell Broadband Engine Architecture, Part 5: Managing Memory, (2006) http://www-128.ibm.com/developerworks/edu/pa-dw-pa-cbecompile5-i.html
4. Cell Broadband Engine Programming Handbook, Version 1.0, April (2006) http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/9F820A5FFA3ECE8C8725716A0062585F/$file/BE_Handbook_v1.0_10May2006.pdf
5. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI – The Complete Reference, Volume 1, The MPI Core, second edition, MIT Press (1998)
6. Buntinas, D., Mercier, G., Gropp, W.: Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. Proceedings of the Euro PVM/MPI Conference, (2006)
7. Gropp, W., Lusk, E.,: Reproducible Measurements of MPI Performance Characteristics, Argonne National Lab Technical Report ANL/MCS/CP-99345, (1999)
8. Buntinas, D., Mercier, G., Gropp, W.: 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)
9. Buntinas, D., Mercier, G., Gropp, W.: 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
10. Tang, H., Shen, K., Yang, T.: Program Transformation and Runtime Support for Threaded MPI Execution on Shared-Memory Machines, ACM Transactions on Programming Languages and Systems, 22 (2000) 673-700
11. Gropp, W., Lusk, E.,: A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer, Parallel Computing, 22 (1997) 1513-1526
12. Fatahalian, K., Knight, T.J., Houston, M., Erez, M.,: Sequoia: Programming the Memory Hierarchy, Proceedings of SC2006, (2006)
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. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI Microtask for Programming the Cell Broadband Engine^{TM} Processor, IBM Systems Journal, 45 (2006) 85-102