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 University, Prashanthi Nilayam, India
{kris.velamati, arunkumar.thondapu, nareshjs, rgskumar1983, baruahpk}@gmail.com, rrs_parthi@rediffmail.com
[2]IBM, Austin
skapoor@us.ibm.com
[3]Dept. of Computer Science, Florida State University
asriniva@cs.fsu.edu

Abstract. The Cell Broadband Engine shows much promise in high performance computing applications. The Cell is a heterogeneous multi-core processor, with the bulk of the computational work load meant to be borne by eight co-processors called SPEs. Each SPE operates on a distinct 256 KB local store, and all the SPEs also have access to a shared 512 MB to 2 GB main memory through DMA. The unconventional architecture of the SPEs, and in particular their small local store, creates some programming challenges. We have provided an implementation of core features of MPI for the Cell to help deal with this. This implementation views each SPE as a node for an MPI process, with the local store used as if it were a cache. In this paper, we describe synchronous mode communication in our implementation, using the rendezvous protocol, which makes MPI communication for long messages efficient. 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. This demonstrates that it is possible to efficiently implement MPI calls even on the simple SPE cores.

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 [15]. 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 *intra-Cell* MPI 1 implementation that uses each SPE as if it were a node for an MPI process [16, 17].

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, which have recently become available. We have addressed the others in [16] and in this paper, related to the MPI implementation. While [16] focuses on buffered mode communication, this paper focuses on synchronous mode communication. These two modes are explained in greater detail below.

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. These features are available in the latest release of the Cell SDK. These will allow applications to be ported in a generic manner, with minimal changes to the code, when used along with our MPI implementation. Meanwhile, in order to evaluate the performance of our implementation, we hand-coded these transformations for two applications with large data, before the SDK features were made available. We have also developed a version of our MPI implementation for small memory applications, which can be ported directly, maintaining application data in local store. However, we will primarily describe our implementation that uses the local store as a software controlled cache, providing additional information on the small memory implementation occasionally.

Empirical results show that our synchronous mode implementation achieves good performance, with throughput as high as *6.01* GB/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 Sect. 2, we describe the architectural features of Cell that are relevant to the MPI implementation. We then describe our implementation in Sect. 3 and evaluate its performance in Sect. 4. We finally summarize our conclusions in Sect. 5.

2. Cell Architecture

Fig. 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.

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.

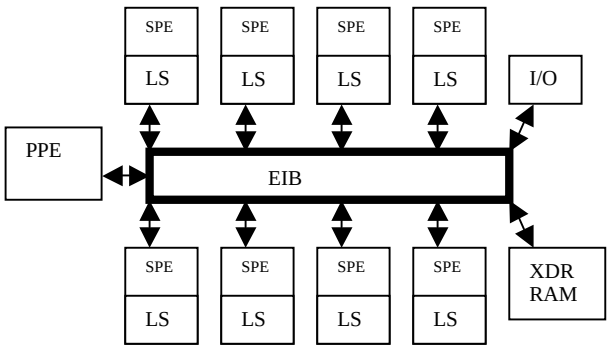
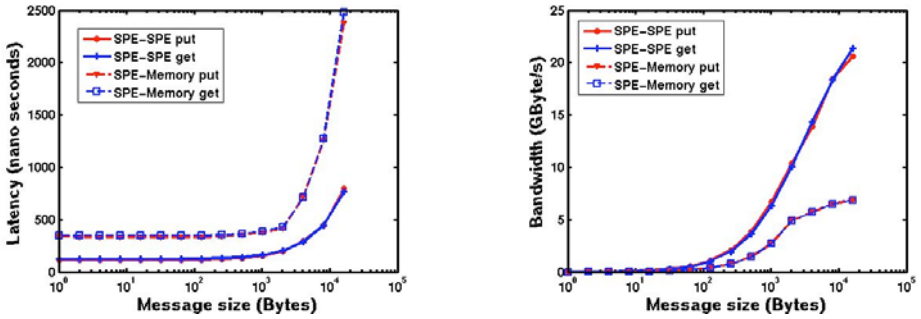


Fig. 1. Overview of cell architecture

In order to use the SPEs, a process running on the PPE can spawn threads that run on the SPEs. A SPE's local store and registers are mapped onto the effective address of the process that spawned a 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 latter attain a maximum bandwidth of around 7 GB/s, in contrast to over 20 GB/s for the former. The latencies are a little higher when multiple SPEs simultaneously access memory.



(a) (b)

Fig. 2. Latency and bandwidth of DMA operations

3. MPI Design

In this section, we describe our basic design for synchronous mode 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. Implementations can 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] [13].

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. In this paper, we describe only our synchronous mode implementation, and present experimental results where this mode is used for all message sizes[2].

---

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

[2] We described the buffered mode in [16]. It has smaller short-message latency but has poorer performance for long messages. The experimental results in [16] show the performance of a

3.2 MPI Initialization

We first summarize the MPI initialization process presented in [16]. 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. In fact, an earlier implementation, which used some helper threads on the PPE to assist with communication, showed poor performance.

3.3 Synchronous mode 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

---

hybrid implementation, which switches to synchronous mode for large messages. However, the synchronous mode implementation is not described in [16].

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

[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.

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. We describe the protocol for data in contiguous locations, which is the more common case; data in non-contiguous locations will incur additional overhead. The send operation from P_i to P_j proceeds as follows. The send operation first puts the meta-data entry into buffer B_{ij} through a DMA operation.

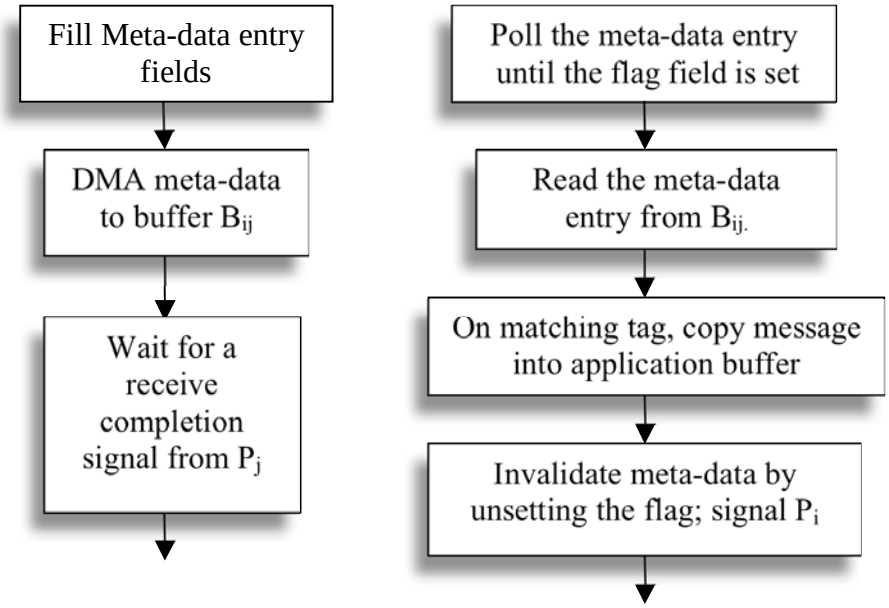


Fig. 3. Execution of (a) send and (b) receive operations from a specific source SPE P_i to P_j

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. Fig. 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.

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, and so we perform this check[5]. If the tag matched, then the address of the message is obtained from the meta-data entry.

The receive call then transfers data from the source SPE's application buffer to its own 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 [10]; *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 block copy commands to avoid at least one of these misses. But these cache misses still 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.

4. 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 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.

---

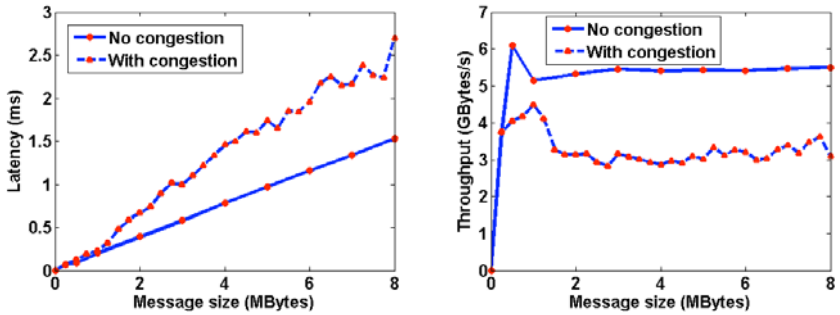
[5] We do not mention other error checks that are fairly obvious.

[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.

Fig. 4 shows the latency and bandwidth results using the pingpong test from *mpptest* [9]. 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 [9]. 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.



(a) (b)

Fig. 4. Latency and throughput results for point-to-point communication

Fig. 4 (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).

Fig. 4 (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 7 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 for the main memory. Thus the observed DMA time agrees with what one would expect from the DMA times.

Table 1. Latency and bandwidth comparison

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

Table 1 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 [14]. *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. Conclusions

We have described an efficient implementation of synchronous mode MPI communication on the Cell processor. It is a part of an MPI implementation that demonstrates 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 directly, without any changes to the code being required. Other applications can make relatively small hand-coded changes, along with features provided by the SDK. 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 University, 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. Buntinas, D., Mercier, G., Gropp, W.: Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem, In Proceedings of the Euro PVM/MPI Conference, (2006)
4. Buntinas, D., Mercier, G., Gropp, W.: Data Transfers Between Processes in an SMP System: Performance Study and Application to MPI, In Proceedings of the International Conference on Parallel Processing, (2006) 487–496
5. Buntinas, D., Mercier, G., Gropp, W.: Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem, In 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. Fatahalian, K., Knight, T.J., Houston, M., Erez, M.: Sequoia: Programming the Memory Hierarchy, In Proceedings of SC2006, (2006)
8. Gropp, W., Lusk, E.: A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer, Parallel Computing, 22 (1997) 1513–1526
9. Gropp, W., Lusk, E.: Reproducible Measurements of MPI Performance Characteristics, Argonne National Lab Technical Report ANL/MCS/CP-99345, (1999)
10. Jin, H.-W., Panda, D.K.: LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster, In Proceedings of the International Conference on Parallel Processing, (2005) 184–191
11. 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

12. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI Microtask for Programming the Cell Broadband Engine™ Processor, IBM Systems Journal, 45 (2006) 85–102
13. 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)
14. 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
15. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The Potential of the Cell Processor for Scientific Computing, In Proceedings of the ACM International Conference on Computing Frontiers, (2006).
16. Krishna, M., Kumar, A., Jayam, N., Senthilkumar, G., Baruah, P.K., Sharma, R., Srinivasan, A., Kapoor, S.: 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)
17. Krishna, M., Kumar, A., Jayam, N., Senthilkumar, G., Baruah, P.K., Sharma, R., Srinivasan, A., Kapoor, S.: Brief Announcement: Feasibility Study of MPI Implementation on the Heterogeneous Multi-Core Cell BE™ Architecture, to appear in Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), (2007)