Optimizing Assignment of Threads to SPEs on the Cell BE Processor

C.D. Sudheer, T. Nagaraju, and P.K. Baruah
Dept. of Mathematics and Computer Science
Sri Sathya Sai University
Prashanthi Nilayam, India
sudheer@sssu.edu.in

Ashok Srinivasan
Dept. of Computer Science
Florida State University
Tallahassee, USA
asriniva@cs.fsu.edu

Abstract

The Cell is a heterogeneous multicore processor that has attracted much attention in the HPC community. The bulk of the computational workload on the Cell processor is carried by eight co-processors called SPEs. The SPEs are connected to each other and to main memory by a high speed bus called the Element Interconnect Bus (EIB), which is capable of 204.8 GB/s. However, access to the main memory is limited by the performance of the Memory Interface Controller (MIC) to 25.6 GB/s. It is, therefore, advantageous for the algorithms to be structured such that SPEs communicate directly between themselves over the EIB, and make less use of memory. We show that the actual bandwidth obtained for inter-SPE communication is strongly influenced by the assignment of threads to SPEs (thread-SPE affinity) in many realistic communication patterns. We identify the bottlenecks to optimal performance and use this information to determine good affinities for common communication patterns. Our solutions improve performance by up to a factor of two over the default assignment. We also discuss the optimization of affinity on a Cell blade consisting of two Cell processors, and provide a software tool to help with this. Our results will help Cell application developers choose good affinities for their applications.

1. Introduction

The Cell is a heterogeneous multi-core processor that has attracted much attention in the HPC community. It contains a PowerPC core, called the PPE, and eight co-processors, called SPEs. The SPEs are meant to handle the bulk of the computational workload, and have a combined peak speed of 204.8 Gflop/s in single precision and 14.64 Gflop/s in double precision. They are connected to each other and to main memory by a high speed bus called the EIB, which has a bandwidth of 204.8 GB/s.

However, access to main memory is limited by the memory interface controller's performance to 25.6 GB/s total (both directions combined). If all eight SPEs access main memory simultaneously, then each sustains bandwidth less than 4 GB/s. On the other hand, each SPE is capable of simultaneously sending and receiving data at 25.6 GB/s in each direction. Latency for inter-SPE communication is under 100 ns for short messages, while it is a factor of two greater to main memory. It is, therefore, advantageous for algorithms to be structured such that SPEs tend to communicate more between themselves, and make less use of main memory.

The latency between each pair of SPEs is identical for short messages and so affinity does not matter in this case. In the absence of contention for the EIB, the bandwidth between each of pair of SPEs is identical for long messages too, and reaches the theoretical limit. However, we show later that in the presence of contention, the bandwidth can fall well short of the theoretical limit, even when the EIB's bandwidth is not saturated. This happens when the message size is greater than 16 KB. It is, therefore, important to assign threads to SPEs to avoid contention, in order to maximize the bandwidth for the communication pattern of the application.

We first identify causes for the loss in performance, and use this information to develop good thread-SPE affinity schemes for common communication patterns, such as ring, binomial-tree, and recursive doubling. We show that our schemes can improve performance by over a factor of two over a poor choice of assignments. By default, the assignment scheme provided is somewhat random, which sometimes leads to poor affinities and sometimes to good ones. With many communication patterns, our schemes yield performance that is close to twice as good as the average performance of the default scheme. Our schemes also lead to more predictable performance, in the sense that the standard deviation of the bandwidth obtained is lower.

We also discuss optimization of affinity on a Cell blade consisting of two Cell processors. We show that the affinity within each processor is often less important than the assignment of threads to processors. We provide a software tool that partitions the threads amongst the two processors, to yield good performance.

The outline of the rest of the paper is as follows. In § 2, we summarize important architectural features of the Cell processor relevant to this paper. We next show that thread-SPE affinity can have significant influence on inter-

SPE communication patterns in § 3. We also identify factors responsible for reduced performance. We use these results to suggest good affinities for common communication patterns. We then evaluate them empirically in § 4. We next discuss optimizing affinity on the Cell blade in § 5. We finally present our conclusions in § 6. More details on our results, and on related work, are presented in a technical report [3].

2. Cell Architecture Overview

We summarize below the architectural features of the Cell of relevance to this work, concentrating on the communication architecture. Further details can be found in [2], [5].

Figure 1 provides an overview of the Cell processor. It contains a cache-coherent PowerPC core called the PPE, and eight co-processors, called SPEs, running at 3.2 GHz each. It has a 512 MB - 2 GB external main memory. An XDR memory controller provides access to main memory at 25.6 GB/s total, in both directions combined. The PPE, SPE, and memory controller are connected via the EIB. The maximum bandwidth of the EIB is 204.8 GB/s. In a Cell blade, two Cell processors communicate over a BIF bus. The numbering of SPEs on processor 1 is similar, except that we add 8 to the rank for each SPE.

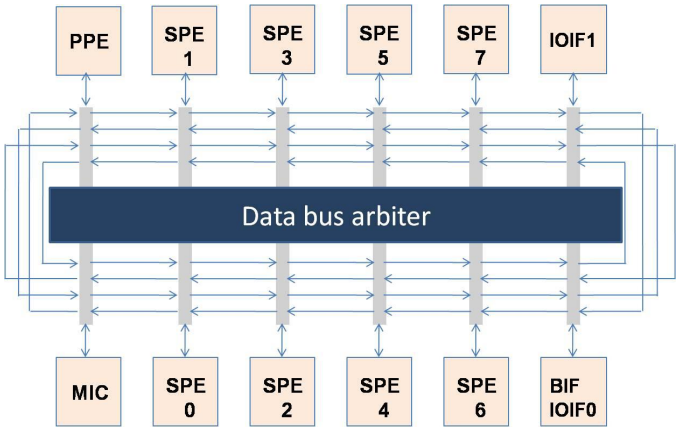

Figure 1. Overview of the Cell communication architecture.

The SPEs have only 256 KB local store each, and they can directly operate only on this. They have to explicitly fetch data from memory through DMA in order to use it. Each SPE can have 16 outstanding requests in its DMA queue. Each DMA can be for at most 16 KB. However, a DMA list command can be used to scatter or gather a larger amount of data. When we perform experiments on messages larger than 16 KB, we make multiple non-blocking DMA requests, with total size equal to the desired message size, and then wait for all of them to complete.

In order to use an SPE, a process running on the PPE spawns an SPE thread. Each SPE can run one thread, and that thread accesses data from its local store. The SPEs' local stores and registers are mapped to the effective address space of the process that spawned the SPE threads. SPEs can use these effective addresses to DMA data from or to another SPE. As mentioned earlier, data can be transferred much faster between SPEs than between SPE and main memory [2], [5].

The data transfer time between each pair of SPEs is independent of the positions of the SPEs, if there is no other communication taking place simultaneously [3]. However, when many simultaneous messages are being transferred, transfers to certain SPEs may not yield optimal bandwidth, even when the EIB has sufficient bandwidth available to accommodate all messages.

In order to explain this phenomenon, we now present further details on the EIB. The EIB contains four rings, two running clockwise and two running counter-clockwise. All rings have identical bandwidths. Each ring can simultaneously support three data transfers, provided that the paths for these transfers don't overlap. The EIB data bus arbiter handles a data transfer request and assigns a suitable ring to a request. When a message is transferred between two SPEs, the arbiter provides it a ring in the direction of the shortest path. For example, transfer of data from SPE 1 to SPE 5 would take a ring that goes clockwise, while a transfer from SPE 4 to SPE 5 would use a ring that goes counter-clockwise. If the distances in clockwise and anti-clockwise directions are identical, then the message can take either direction, which may not necessarily be the best direction to take, in the presence of contention. From these details of the EIB, we can expect that certain combinations of affinity and communication patterns can cause non-optimal utilization of the EIB. We will study this in greater detail below.

3. Influence of Thread-SPE Affinity on Inter-SPE Communication Performance

In this section, we show that the affinity significantly influences communication performance when there is contention. We then identify factors that lead to loss in performance, which in turn enables us to develop good affinity schemes for a specified communication pattern.

Experimental Setup

The experiments were performed on the CellBuzz cluster at the Georgia Tech STI Center for Competence for the Cell BE. It consists of Cell BE QS20 dual-Cell blades running at 3.2 GHz with 512 MB memory per processor. The codes were compiled with the ppuxlc and spuxlc compilers, using the -O3 -qstrict flags. SDK 2.1 was used. Timing was performed by using the decrementer register which runs at 14.318 MHz, yielding a granularity of around 70 ns. Further details are provided in [3].

Influence of Affinity

We mentioned earlier that affinity does not matter when there is no contention. Figure 2 presents performance results when there are several messages simultaneously in transit, for the following communication pattern: threads *ranked* i and $i + 1$ exchange data with each other, for even i. This is a common communication pattern, occurring in the first phase of recursive doubling algorithms, which are used in a variety of collective communication operations. The results are presented for three specific affinities and for the default. (The default affinity is somewhat random as mentioned earlier; we present results for one specific affinity returned by default – a more detailed analysis is presented later.) We provide details on the affinities used in a later section (§ 4). Our aim in this part of the section is just to show that affinity influences performance.

Figure 2 is meant to identify the message size at which affinity starts to matter. Note that the bandwidths obtained by different SPEs differ. The results show the minimum of those bandwidths. This is an important metric because, typically, the speed of the whole application is limited by the performance of the slowest processor. We can see that the affinity does not matter at less than 16 KB data size, because the network bandwidth is not fully utilized then. With large messages, as a greater fraction of the the network bandwidth is utilized, affinity starts having a significant effect. Figure 3 shows the bandwidth obtained by each SPE with two different affinities, in one particular trial. We can see that some SPEs get good performance, while others perform poorly with a bad affinity.

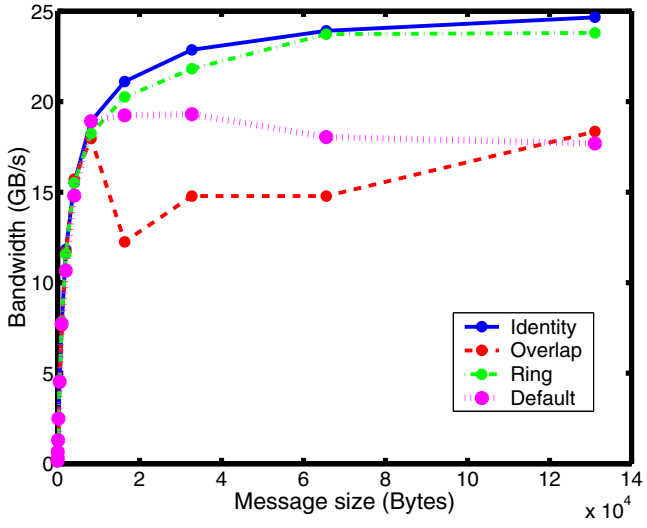

Figure 2. Performance in the first phase of recursive doubling – minimum bandwidth versus message size.

We next give statistics from several trials. Figure 4 gives the results for the default affinity. If we compare it with

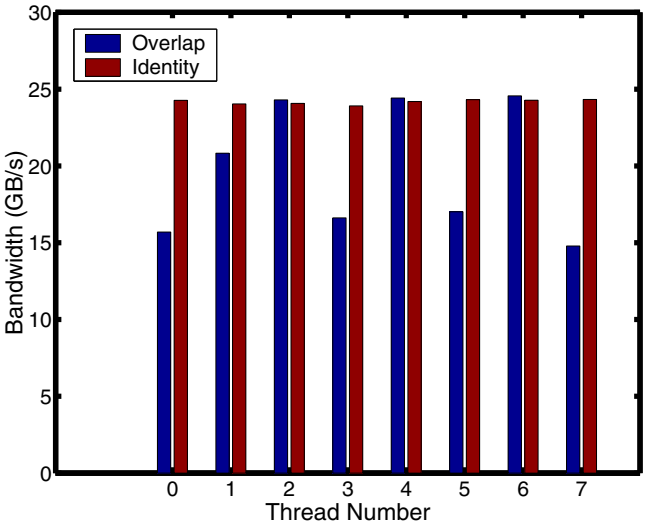

Figure 3. Performance in the first phase of recursive doubling – bandwidth on different SPEs for messages of size 64 KB.

fig. 6, we can see that the standard deviation is much higher for the default affinity than for the "Identity" affinity (which is described later). There are two possible reasons for this. (i) In different trials, different affinities are produced by default, which can lead to a large variance. (ii) For the same affinity, the variance could be high. It turns out that in these eight trials, only two different affinities were produced. The mean and standard deviation for each affinity was roughly the same. The high variance is primarily caused by the inherent variability in each affinity. This often happens in cases where contention degrades performance – when there is much contention, there is more randomness in performance which causes some SPEs to be have lower performance. Figure 4 shows that most of the SPEs have good mean performance. However, we also observe that most SPEs have a low value of their worst performance. In most trials, some thread or the other has poor performance, which proves to be a bottleneck. In contrast, all SPEs consistently obtain good performance with the Identity affinity. This contrast is illustrated in fig. 5 and fig. 7. For some other communication patterns, the default assignment yields some affinities that give good performance and some that yield poor performance. In those cases the variance is more due to the difference in affinities.

Performance Bottlenecks

We experimented with simple communication steps, in order to identify factors that are responsible for loss in performance. These results, which are presented in [3], suggest the following rules of thumb for large messages.

1) Avoid overlapping paths for more than two messages in the same direction. This is the most important

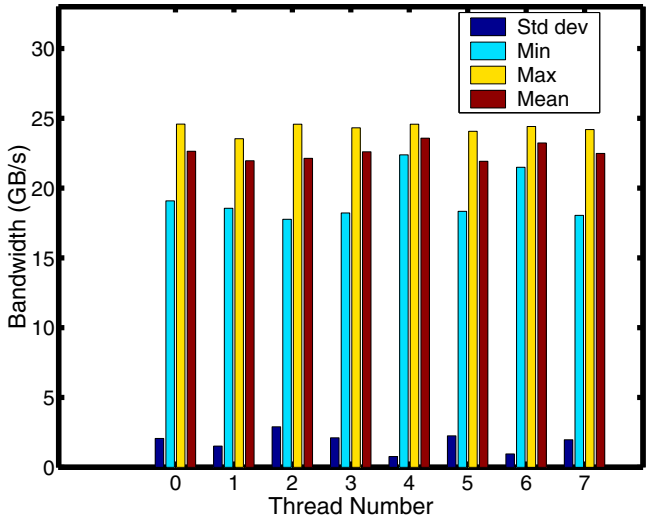

Figure 4. Performance in the first phase of recursive doubling with default affinities – bandwidth on each thread.

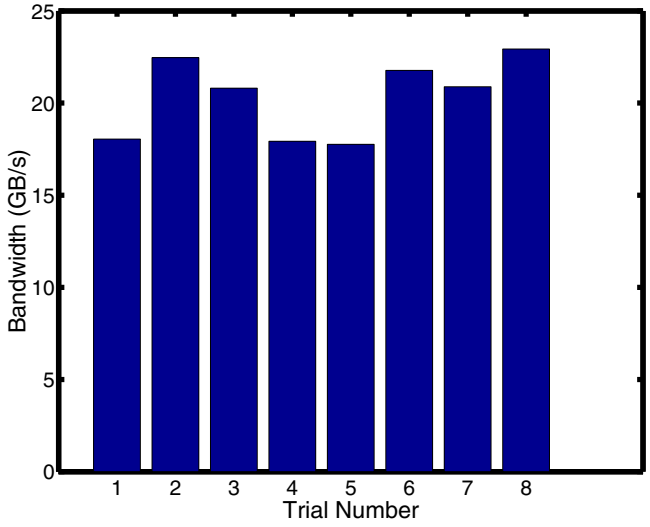

Figure 5. Performance in the first phase of recursive doubling with default affinities – minimum bandwidth in each of the eight trials.

observation.

2) Given the above constraints, minimize the number of messages in any direction by balancing the number of messages in both directions.
3) Don't make any assumptions regarding the direction of transfer for messages that travel half-way across the EIB ring.

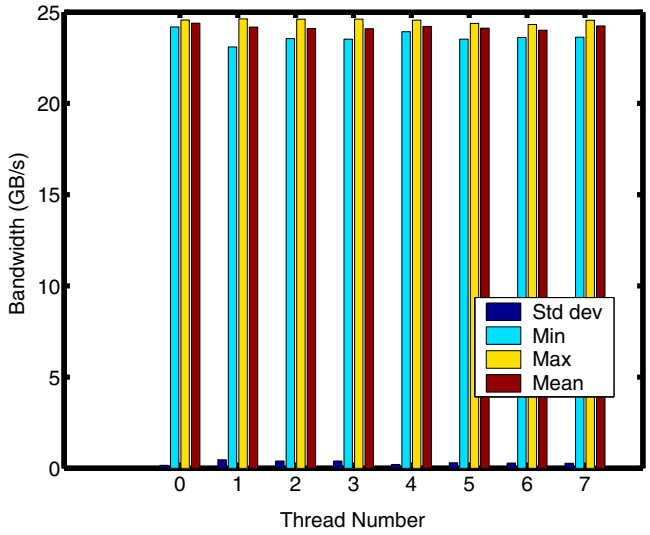

Figure 6. Performance in the first phase of recursive doubling with the "Identity" affinity – bandwidth on each thread.

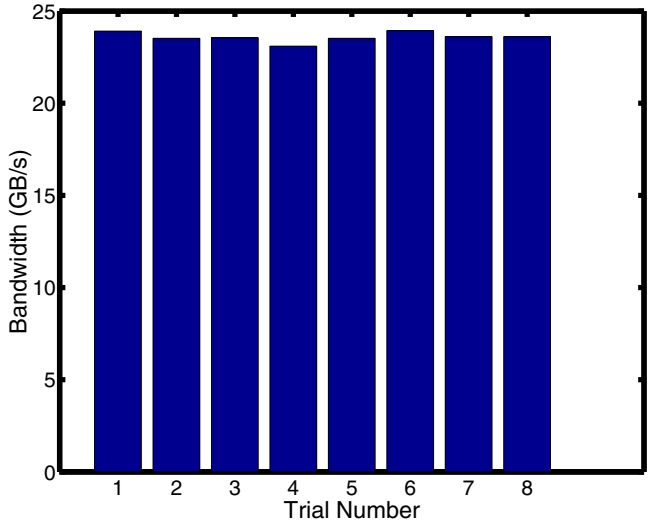

Figure 7. Performance in the first phase of recursive doubling with the "Identity" affinity – minimum bandwidth in each of the eight trials.

4. Affinities and Their Evaluation

We used the following affinities, other than the default. They were designed to avoid the bottlenecks mentioned in § 3.

- *Identity* The thread ID is identical to the physical ID of the SPE.
- *Ring* (Physical ID, Thread Number) mapping: {(0, 0), (1, 7), (2, 1), (3, 6), (4, 2), (5, 5), (6, 3), (7, 4)}. Thread ranks that are adjacent are also physically

Pattern	Name
1. 0 ← 7, 1 ← 0, 2 ← 1, ..., 7 ← 6	Ring
2. 0 ↔ 1, 2 ↔ 3, 4 ↔ 5, 6 ↔ 7	Recursive doubling 1
3. 0 ↔ 2, 1 ↔ 3, 4 ↔ 6, 5 ↔ 7	Recursive doubling 2
4. 0 ↔ 4, 1 ↔ 5, 2 ↔ 6, 3 ↔ 7	Recursive doubling 3
5. 0 ← 2, 1 ← 3, 2 ← 4, 3 ← 5 4 ← 6, 5 ← 7, 6 ← 0, 7 ← 1,	Bruck 2
6. 1 ← 0, 3 ← 2, 5 ← 4, 7 ← 6	Binomial-tree 3

Table 1. Communication patterns studied. The numbers in the first column refer to thread ranks. The numbers in the second column give the phase of the corresponding algorithm.

adjacent on the EIB ring, and thread 7 is physically adjacent to thread 0.

- *Overlap* Mapping: {(0, 0), (1, 7), (2, 2), (3, 5), (4, 4), (5, 3), (6, 6), (7, 1)}. Here, threads with adjacent ranks are half way across the ring. We would expect poor results on a ring communication pattern. We use this as a lower bound on the performance for a ring communication pattern.
- *EvenOdd* Mapping: {(0, 0), (1, 4), (2, 2), (3, 6), (4, 1), (5, 5), (6, 3), (7, 7)}. Even ranks are on the left hand side and odd ranks are on the right hand side. This affinity was designed to perform well with recursive doubling.
- *Leap2* Mapping: {(0, 0), (1, 4), (2, 7), (3, 3), (4, 1), (5, 5), (6, 6), (7, 2)}. This deals with a limitation of the Ring affinity on the ring communication pattern. The Ring affinity causes all communication to be in the same direction with that communication pattern, causing unbalanced load. The Leap2 affinity causes adjacent ranks to be two apart in the sequence. This leads to balanced communication in both directions with the ring pattern.

We consider the communication patterns specified in table 1. We also considered a few communication patterns that have multiple phases, whose results are reported in [3].

Experimental Results

Figures 8 - 13 show the performance of the different affinity schemes with the communication patterns mentioned in table 1. *In these results, we report the performance of the message with the lowest bandwidth, because this will typically be the bottleneck for an application.*

Figure 8 shows results with the ring communication pattern. We can see that the Overlap affinity gets less than half the performance of the best patterns. This is not surprising, because this affinity was developed to give a lower bound on the performance on the ring pattern, with a large number of overlaps. The Ring affinity does not have any overlap, but it has a very unbalanced load, with all the transfers going counter-clockwise, leading to poor performance. The Leap2 pattern does not have any overlapping paths in each direction, and so it gives good performance. The Identity affinity has only one explicit overlap in each direction, which by itself does not degrade performance. However, it also has a few paths that go half-way across the ring, and these cause additional overlap whichever way they go, leading to loss in performance. The EvenOdd affinity has a somewhat similar property; however, the paths that go half-way across have a direction in which they do not cause an overlap. It appears that these good paths are taken, and so the performance is good.

The default affinity is somewhat random. So, we shall not explicitly give reasons for its performance below. We show results with the default affinity primarily to demonstrate the improvement in performance that can be obtained by explicitly specifying the affinity. The randomness in this affinity also leads to a higher standard deviation than for the other affinities.

Figure 9 shows results with the first phase of recursive doubling. The Overlap pattern has all paths going half-way across. So, there is extensive overlap, and consequently poor performance. The other four deterministic patterns yield good performance. The Ring affinity has no overlap in each direction. The other three have overlaps; however, the transfers can be assigned to distinct rings on the EIB such that there is no overlap in each ring, leading to good performance.

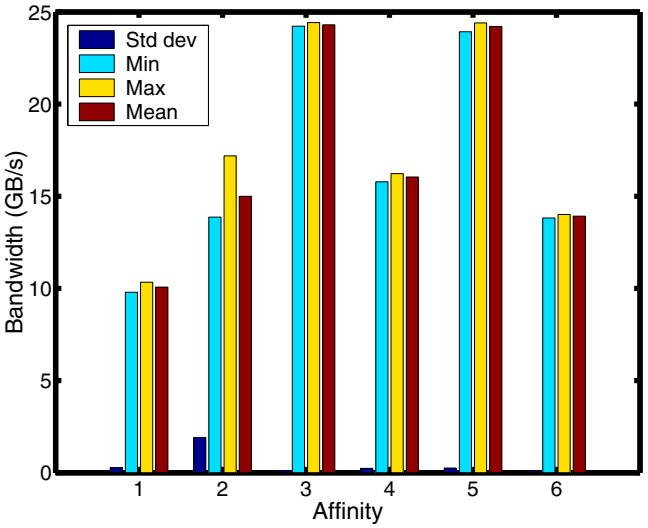

Figure 8. Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring – ring pattern.

Figure 10 shows results with the second phase of recursive doubling. Leap2 alone has poor performance, because it has all paths going half-way across. The Ring affinity has overlaps that can be placed on different rings. The other

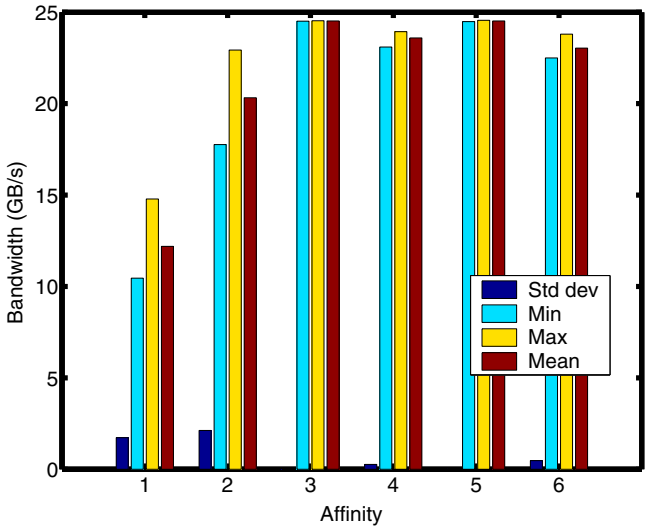

Figure 9. Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring – first phase of recursive doubling.

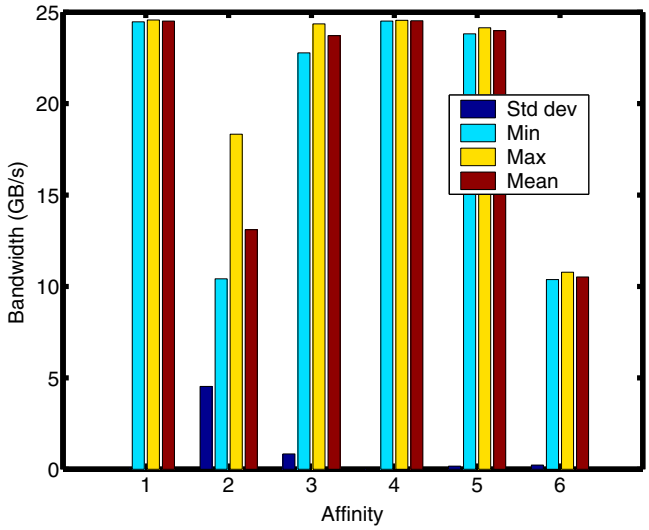

Figure 11. Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring – third phase of recursive doubling.

deterministic patterns do not have any overlap in the same direction. The third phase of recursive doubling is shown in fig. 11. The Ring affinity alone has poor performance, among the deterministic affinities, because it has all paths going half-way across. The other deterministic patterns have overlaps that can be placed on distinct rings, leading to good performance.

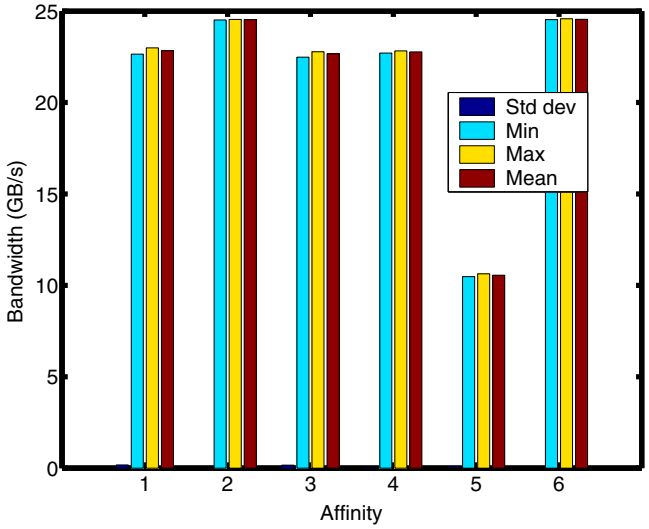

Figure 10. Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring – second phase of recursive doubling.

Figure 12 shows results for the second phase of the Bruck algorithm[1]. The EvenOdd affinity performs best, because it does not have any overlap in the same direction. Identity too does not have any overlap, and gives good performance. However, its performance is a little below that of EvenOdd. The Ring affinity has all transfers going in the same direction and gets poor performance. The Overlap affinity has no overlap, but has unbalanced load, with the counter-clockwise ring handling six of the eight messages, which reduces its performance. The Leap2 affinity has several transfers going half-way across, which result in overlap with either choice of direction, and consequently lead to poor performance.

All affinities perform well on the third phase of binomial-tree[2], shown in fig. 13. In the Ring affinity, all messages go in the same direction. However, since there are only four messages in total, the performance is still good. All the other affinities have two transfers in each direction, and each of these can be placed on a distinct ring, yielding good performance.

We next consider the performance of the above affinity schemes in a practical application. This is a Monte Carlo application for particle transport, which tracks a number of random walkers on each SPE [4]. Each random walker takes a number of steps. A random walker may be terminated on occasion, causing a load imbalance. In order to keep the load balanced, we use the diffusion scheme, in which a certain fraction of the random walkers on each SPE is

1. The first phase of the Bruck algorithm is identical to the ring pattern, and the third phase is identical to the third phase of recursive doubling, and they are, therefore, not presented separately.

2. The first phase of binomial-tree has only one message, and the second phase has only two messages. Therefore, there is no contention in these phases.

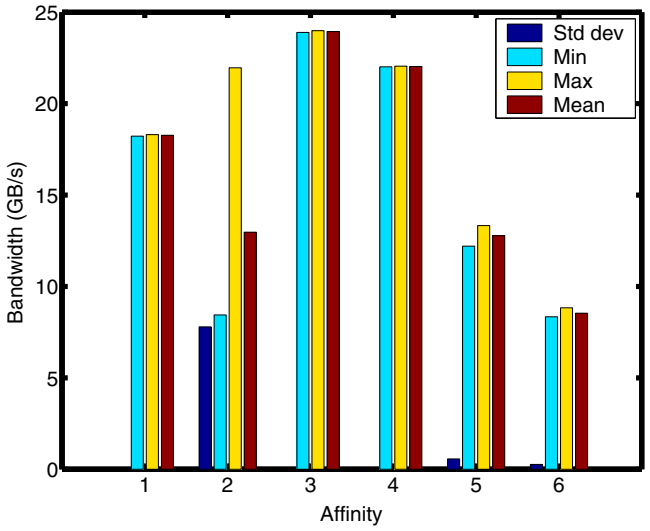

Figure 12. Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring – second phase of Bruck algorithm for all-gather. Right: Third phase of binomial-tree.

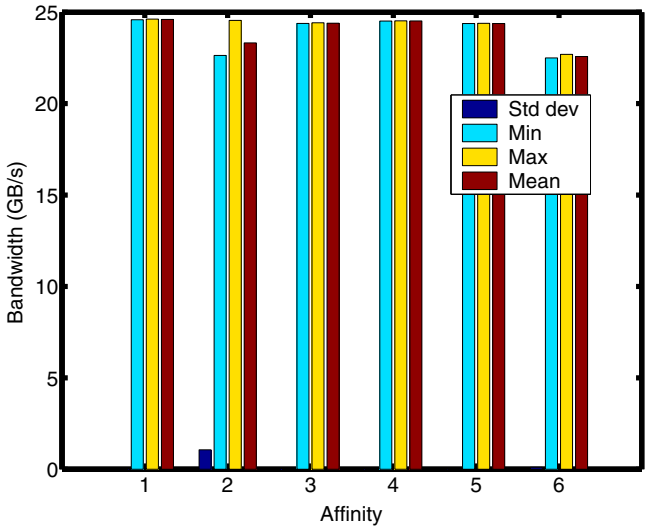

Figure 13. Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring – third phase of binomial-tree.

transfered to a neighbor after each step. SPE i considers SPEs $i-1$ (modulo N) and $i+1$ (modulo N) as its neighbors when we use N SPEs. We can see a factor of two difference between the communication costs for the best and worst affinities in fig. 14. Figure 15 shows a difference in total application performance of over 10% between the best and worst affinities.

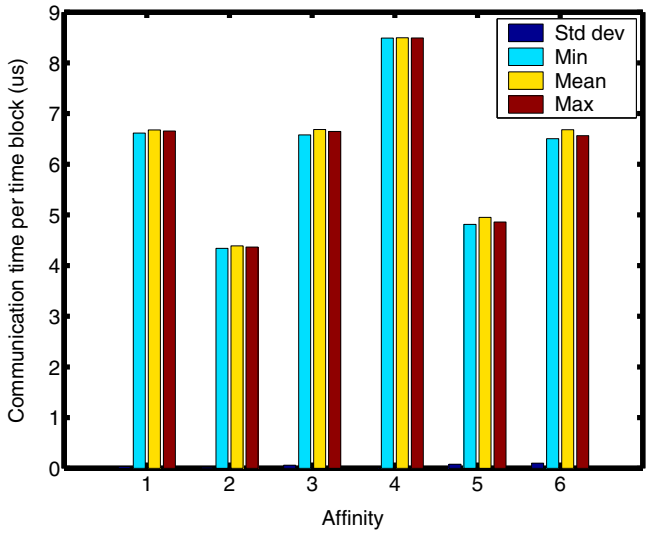

Figure 14. Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring, when used in a particle transport application – communication time.

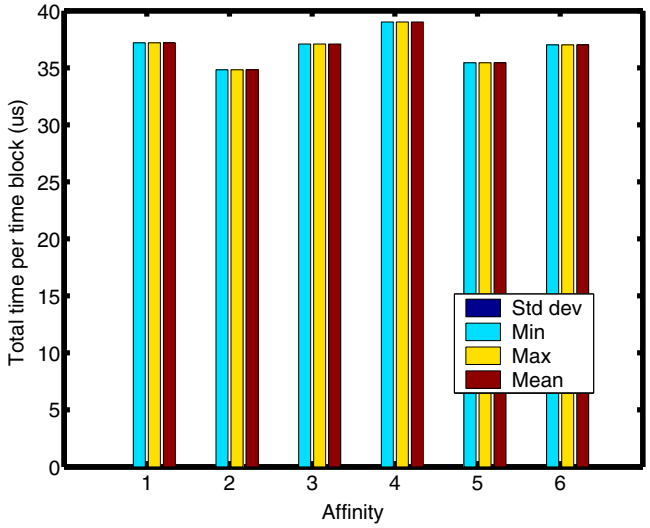

Figure 15. Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring, when used in a particle transport application – total application time.

5. Affinity on a Cell Blade

Communication on a Cell blade is asymmetric, with around 30 GB/s theoretically possible from Cell 0 to Cell 1, and around 20 GB/s from Cell 1 to Cell 0. However, as shown in table 2, communication between a single pair of SPEs on different processors of a blade yields bandwidth

Processor 1 to 0	Processor 0 to 1	Bandwidth 1 to 0	Bandwidth 0 to 1
1	0	3.8 GB/s	
0	1		5.9 GB/s
1	1	3.8 GB/s	5.9 GB/s
2	2	3.4 GB/s	5.0 GB/s
3	3	2.8 GB/s	3.6 GB/s
4	4	2.6 GB/s	2.7 GB/s
5	5	2.0 GB/s	2.0 GB/s
6	6	1.6 GB/s	1.6 GB/s
7	7	1.3 GB/s	1.3 GB/s
8	8	1.2 GB/s	1.2 GB/s

Table 2. Performance of each message for different numbers of SPEs communicating simultaneously across processors on a Cell blade with 64 KB messages each.

Affinity	Bandwidth
Identity	3.8 GB/s
EvenOdd	3.5 GB/s
Ring	3.9 GB/s
Worst case	1.3 GB/s

Table 3. Performance for different affinities on a Cell blade (16 SPEs) for 64 KB messages with the ring communication pattern.

much below this theoretical limit[3]. In fact, this limit is not reached even when multiple SPEs communicate, for messages of size up to 64 KB each.

As shown above, the bandwidth attained by messages between SPEs on different processors is much lower than that between SPEs on the same processor. So, these messages become the bottleneck in the communication. In this case, the affinity within each SPE is not as important as the partitioning of threads amongst the two processors. We have created a software tool that evaluates each possible partitioning of the threads amongst the two processors and chooses the one with the smallest communication volume. This tool takes into account the asymmetry in the bandwidth between the two processors, and so a partition sending more data will be placed on processor 0. Table 3 shows that such partitioning is more important than the affinity within each processor. In that figure, the software mentioned above was used to partition the threads for a ring communication pattern, and certain affinities studied above were used within each processor. One case, however, considered a worst-case partitioning where each transfer is between a pair of SPE on different processors.

6. Conclusions and Future Work

We have shown that the SPE-thread affinity has a significant effect on inter-SPE communication performance, for common communication patterns and also for a real application. Specifying a good affinity can improve performance by a factor of two over using the default assignment for many communication patterns. Furthermore, the performance is more predictable. We have also discussed optimizing affinity on a Cell blade. Here, the assignment of threads to processors is more important than the details of the affinity on each processor. Furthermore, the partitioning of the threads on to the different processors needs to take into account the asymmetry between the communication between the two processors on a Cell blade.

In future work, we wish to develop an algorithm that automatically determines the ideal affinity when given a communication pattern. There are two approaches that we are taking for this. One is to model the communication performance, and then evaluate the performance for each possible affinity. There are 8! possible affinities, which can be reduced to 7!/2 if we use symmetries. The performance of each affinity can easily be evaluated once we develop a model. Yet another approach is to map the communication pattern of the application to a pattern for which we have a good affinity available. For example, the EvenOdd affinity performs well with recursive doubling, and so can be naturally mapped to a hypercube. Many communication patterns, in turn, can be mapped efficiently to a hypercube.

3. This result is consistent with those presented in [1] for the QS21 blade.

Acknowledgments

We acknowledge partial support by NSF grant # DMS-0626180. We thank IBM for providing access to a Cell blade under the VLP program, and to the Cell Center for Competence at Georgia Tech. Finally, we thank Sri Sathya Sai Baba for all his help and inspiration.

References

[1] P. Altevogt, H. Boettiger, T. Kiss, and Z. Krnjajic. IBM blade center QS21 hardware performance. Technical Report WP101245, IBM, 2008.

[2] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26:10–23, 2006.

[3] T. Nagaraju, P.K. Baruah, and A. Srinivasan. Optimizing assignment of threads to SPEs on the Cell BE processor. Technical Report TR-080906, Computer Science, Florida State University, 2008.

[4] G. Okten and A. Srinivasan. Parallel quasi-Monte Carlo methods on a heterogeneous cluster. In *Proceedings of the Fourth International Conference on Monte Carlo and Quasi Monte Carlo (MCQMC)*. Springer-Verlag, 2000.

[5] M.K. Velamati, A. Kumar, N. Jayam, G. Senthilkumar, P.K. Baruah, S. Kapoor, R. Sharma, and A. Srinivasan. Optimization of collective communication in intra-Cell MPI. In *Proceedings of the 14th IEEE International Conference on High Performance Computing (HiPC)*, pages 488–499, 2007.