Optimizing Assignment of Threads to SPEs on the Cell BE Processor

T. Nagaraju* P.K. Baruah† Ashok Srinivasan‡

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 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 (we refer to this as thread-SPE affinity). We first identify the bottlenecks to optimal performance. We 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. The performance is also more predictable. The main contributions of this work are as follows: (i) showing that thread-SPE affinity can significantly influence performance and identifying factors that can lower performance, and (ii) finding a good affinity for common communication patterns.

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 sending and receiving data at 25.6 GB/s (that is, 25.6 GB/s in each

*Mathematics and Computer Science, Sri Sathya Sai University, Prashanthi Nilayam, India, t.nagaraj@gmail.com

†Mathematics and Computer Science, Sri Sathya Sai University, Prashanthi Nilayam, India, baruahpk@gmail.com

‡Computer Science, Florida State University, Tallahassee FL 32306, USA, asriniva@cs.fsu.edu

direction simultaneously). 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 a couple of kilo-bytes. 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[1], 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.

One particular affinity that we developed performs well on all the communication patterns we have studied. For communication patterns different from those we have studied, application developers can use some rules of the thumb we have presented to design a suitable affinity.

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 then mention the experimental setup for our empirical studies in § 3. We next show that thread-SPE affinity can have significant influence on inter-SPE communication patterns in § 4. We also identify factors responsible for reduced performance. We use these results to suggest good affinities for common communication patterns, and evaluate them empirically in § 5. We finally summarize related work in § 6, and present our conclusions in § 7.

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, 6].

Figure 1 (left) 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 Element Interconnect Bus (EIB). The maximum bandwidth of the EIB is 204.8 GB/s.

[1]The default assignment is not deterministic. However, our empirical studies indicate that each of the 8! permutations is not equally likely either.

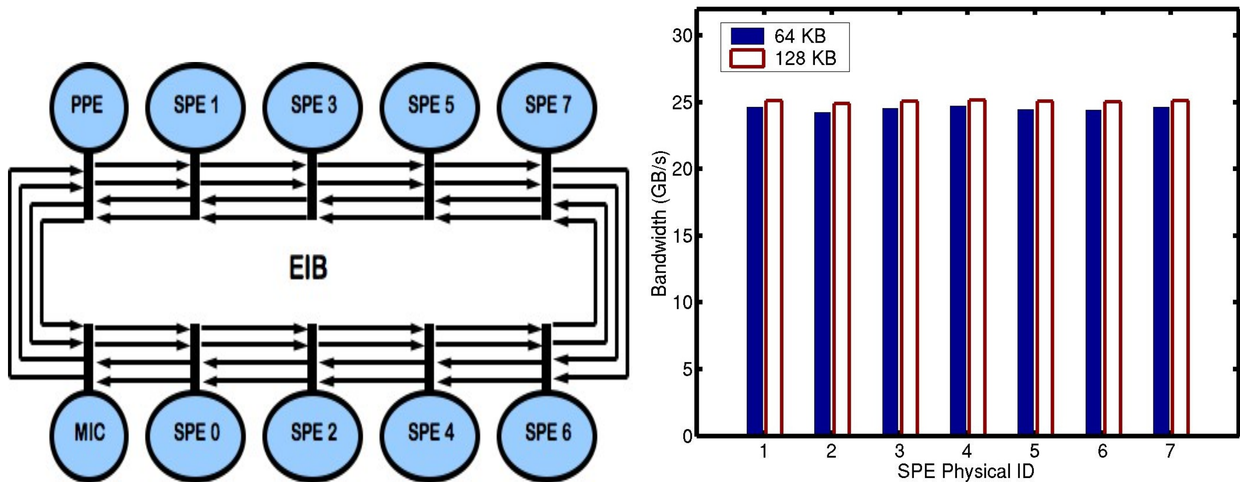

Figure 1: *Left: Overview of the Cell communication architecture. Right: Bandwidth for DMA* get *issued by each SPE for data from SPE 0. Each SPE performs this at different times, and so there is no contention on the EIB.*

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[2]. However, a DMA list command can be used to scatter or gather a larger amount of data.

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 store and registers are mapped to the effective address space of the process that spawned the SPE thread. SPEs can use this effective address 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, 6].

The data transfer time between each pair of SPEs is independent of the position of the SPE, if there is no other communication taking place simultaneously. This is illustrated in fig. 1 (right), which shows the bandwidth when SPE $i, 1 \leq i \leq 7$, DMAs data from the local store of SPE 0. 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 transfer 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. From these details of the EIB, we can expect that certain combinations of affinity and communication

[2]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.

patterns can cause non-optimal utilization on the EIB. We will study this in greater detail below.

3 Experimental Setup

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.

The timing was performed by determining the time taken for 100 calls of the operation to be timed, and dividing by 100. The total time for 100 calls is over a hundred micro-seconds in most cases considered. Therefore, the error due to the granularity of the timing routine is insignificant.

We have implemented a fast barrier operation [6] which takes around 0.4 μs on 8 SPEs. Before starting an operation to be timed, we synchronize the SPEs with a barrier. Otherwise, different SPEs may start at different times, and we may not have as much contention as we expect. In communication patterns with multiple phases, we also call a barrier before starting the timing routine and also before ending the timing. That is, we make the following sequence of calls: Barrier, Start-Timer, Barrier, Repeat-operation-being-timed 100 times, Barrier, End-Timer. The timing on SPE 0 alone is returned for the multi-phase operations, because these patterns typically occur in collective communication, where we wish to time the call over the entire set of SPEs. In other calls, we gather timing results on each SPE that performs a DMA get operation.

Each call was timed either eight times or three times; the former when the variance was high and the latter when the variance was low. The standard deviation computed was the sample standard deviation (normalized by $N-1$ when there are N trials). This gives a slightly higher result than using the population standard deviation. However, it is an unbiased estimate of the standard deviation of the population, unlike the latter. Multiple trials were performed after some delay, so that any short time correlations in the state of the system would not bias the results. This was especially important for the default affinity, were we observed that the affinity obtained tended to be strongly correlated over short periods of time.

Assignment of threads to SPEs

The PPE process for the application always spawned 8 SPE threads using the spe_create_thread call. For the default affinity, we assigned rank i to the i th thread that was created. In different invocations of the application, thread i may be assigned to different SPEs. We used the numactl command to ensure that all SPE threads were created on an SPE on the same processor. The threads run to completion without preemption. We recommend this if affinity matters for the efficiency of an application, in order to ensure that the thread is not migrated to a different SPE.

The spe_create_thread call actually has a parameter to specify the affinity mask. However,

we observed that this mask does not cause the thread to be spawned on the desired SPE. The libspe library also has a spe_set_affinity function available. However, this routine is not actually implemented on our system, even though the call is available. We therefore used the following strategy to specify affinity. A file named /spu/spethread-<ppeid>-<speid>/phys-id contains the physical ID of the SPE on which the specified SPE thread runs. After creating the SPE threads, the PPE process determines the physical ID of each thread. It then maps the physical ID to the desired thread number, as specified in an initialization file that we use in our applications. This thread number is then passed to the SPEs, which wait to receive this information from the PPE before proceeding with the rest of their work. This procedure can be simplified once the standard affinity routines work.

4 Influence of Thread-SPE Affinity on Inter-SPE Communication Performance

In this section, we first 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.

Influence of Affinity

We showed 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 (§ 5). Our aim in this part of the section is just to show that affinity influences performance.

Figure 2 (left) is meant to identify the message size at which affinity starts to matter. Note that the bandwidth obtained by different SPEs differs. 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 2 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 2 (right) 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.

We next give statistics from several trials. Figure 3 gives the results for the default affinity. If we compare it with with fig. 4, 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.

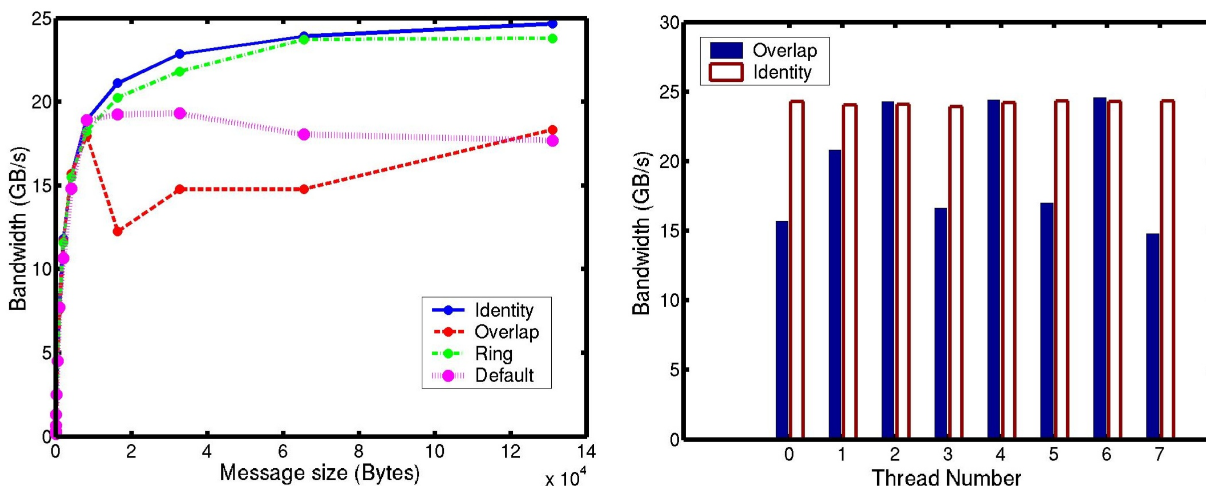

Figure 2: *Performance in the first phase of recursive doubling. Left: Minimum bandwidth versus message size. Right: Bandwidth on different SPEs for messages of size 64 KB.*

The mean and standard deviation for each was roughly the same. The variance is caused primarily due to the inherent variability in each affinity. This often happens in cases were contention degrades performance – when there is much contention, there is more randomness in performance which causes some SPEs to be have lower performance. Figure 3 (left) 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 the right sides of fig. 3 and fig. 4. 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.

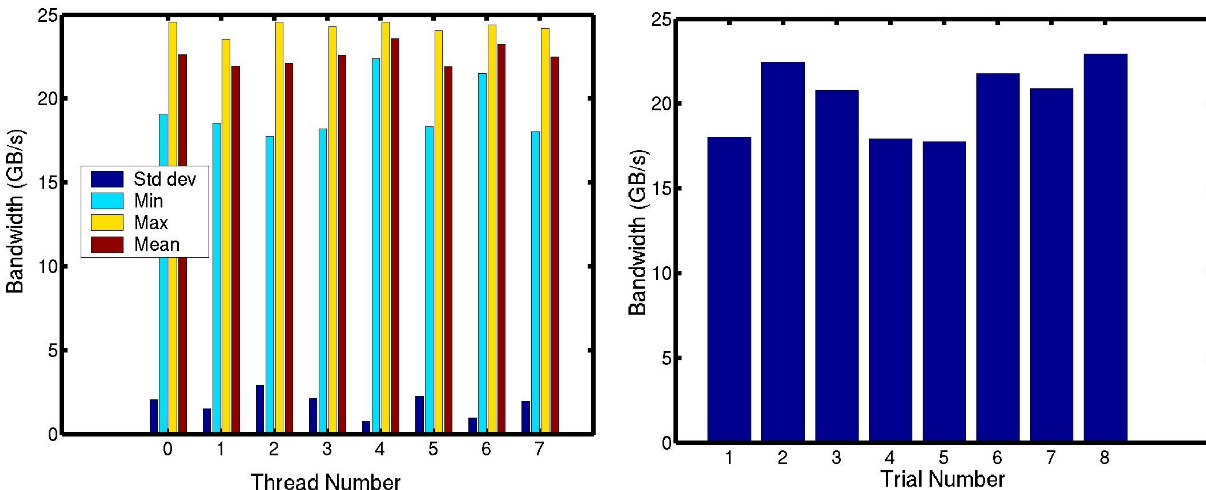

Figure 3: *Performance in the first phase of recursive doubling with default affinities. Left: Bandwidth on each thread. Right: Minimum bandwidth in each of the eight trials.*

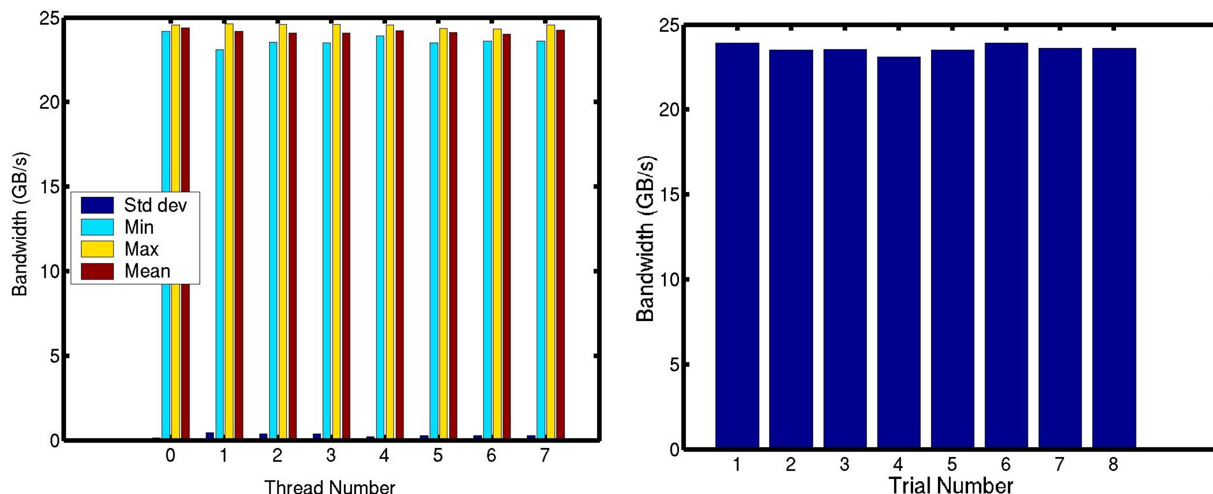

Figure 4: *Performance in the first phase of recursive doubling with the "Identity" affinity. Left: Bandwidth on each thread. Right: Minimum bandwidth in each of the eight trials.*

Performance Bottlenecks

We now experiment with simple communication steps, in order to identify factors that are responsible for loss in performance. In table 1 below, the notation $i \leftarrow j$ means that SPE i gets 64 KB data from SPE j. These numbers refer to the *physical ID* of an SPE, rather than to the thread number. Each result in this table is meant to illustrate some feature. We performed several experiments, but present here only one result for each feature that we evaluated.

The results from table 1 suggest the following rules of thumb when developing a good affinity for a specific communication pattern, where we take into account only large messages.

1. Avoid overlapping paths for more than two messages in the same direction. This is the most important 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[3].

[3]The reasoning behind the conclusions in # 10 in table 1 is as follows. If the transfers that go half-way across the ring can go in only one direction, then (0, 7) and (7, 0) must either be in (i) the same direction, or (ii) in opposite directions. Let us consider case (i) first. In this case, the other three transfers must be in the opposite direction. Otherwise the bandwidth in the direction of (0, 7) is higher than what we expect from the results for three overlapping messages (# 9). But results # 11 are contrary to the other three messages being in the same direction. Thus case (i) is not possible. We now consider case (ii). Either (7, 0) or (0, 7) should be in a direction with just one other transfer, where that other transfer has low bandwidth. However, with two messages, even if there is overlap, good bandwidth can be obtained, as shown in # 8. Therefore, this case is also not possible. This leads to the conclusion that messages that go half-way across the ring can have data transferred in either direction. The unbalanced results also indicate that they are not scheduled optimally to keep the bandwidths balanced. (We want the lowest bandwidth on any transfer to be as high as possible.)

Pattern	**Bandwidth** Mean (Std Dev)	**Comments**
1. 6 ← 4	24.3 (0.04) GB/s	Good bandwidth obtained for a single message.
2. 2 ← 0 6 ← 4	24.2 (0.03) GB/s 24.3 (0.02) GB/s	Good bandwidth obtained with two messages in the same direc- -tion because two rings are available to handle the bandwidth.
3. 2 ← 0 5 ← 7 6 ← 4	23.6 (0.06) GB/s 23.9 (0.01) GB/s 23.8 (0.13) GB/s	Three non-overlapping messages in the same direction. Slight (2-3%) reduction in bandwidth, even though each ring is capable of three simultaneous transfers.
4. 1 ← 3 2 ← 0 5 ← 7 6 ← 4	22.8 (0.17) GB/s 22.9 (0.18) GB/s 22.7 (0.09) GB/s 23.0 (0.17) GB/s	Four non-overlapping messages in the same direction. 5-6% reduction in bandwidth over # 1, even though the two rings together are theoretically capable of handling these at peak bandwidth.
5. 0 ← 1 1 ← 3 2 ← 0 5 ← 7 6 ← 4	19.3 (0.3) GB/s 20.1 (0.4) GB/s 19.3 (0.5) GB/s 19.1 (0.2) GB/s 19.8 (0.4) GB/s	Five non-overlapping messages in the same direction. Around 15% decrease in bandwidth, because the two rings are capable of only 100.4 GB/s aggregate. Simultaneous send and receive on an SPE (SPE 1 here) is not the cause for low performance, as shown in # 6.
6. 2 ← 0 4 ← 2	24.3 (0.02) GB/s 24.7 (0.01) GB/s	Good bandwidth even if one node is both source and destination.
7. 1 ← 3 0 ← 2 5 ← 7 6 ← 4 7 ← 1	22.9 (0.3) GB/s 22.9 (0.3) GB/s 22.7 (0.09) GB/s 22.9 (0.09) GB/s 24.5 (0.06) GB/s	Traffic in one direction (7 ← 1) does not affect performance in the opposite direction. Performance in the counter-clockwise direction is similar to that in #4.
8. 4 ← 2 6 ← 0	24.7 (0.01) GB/s 24.5 (0.02) GB/s	Good bandwidth with two overlapped messages in the same direction, because each ring can handle one.
9. 5 ← 4 6 ← 0 7 ← 2	17.1 (0.4) GB/s 23.4 (0.3) GB/s 18.1 (1.3) GB/s	Bandwidth degrades with three overlapping messages in the same direction. The loss in performance is not uniform. Note the high standard deviation in one case.
10. 0 ← 7 1 ← 6 3 ← 4 5 ← 2 7 ← 0	24.4 (0.1) GB/s 19.0 (0.3) GB/s 17.0 (1.1) GB/s 17.5 (1.0) GB/s 24.3 (0.3) GB/s	Study of messages that travel half-way across the ring. We wish to determine if the messages can take paths in both directions and also if they make effective use of the available bandwidth. The results indicate that they can take paths in both directions, and the performance is not balanced, as explained below.
11 a. 3 ← 4 5 ← 2 6 ← 0 b. 3 ← 4 5 ← 2 0 ← 6	 24.7 (0.01) GB/s 24.6 (0.02) GB/s 24.5 (0.02) GB/s 24.6 (0.02) GB/s 24.6 (0.02) GB/s 24.6 (0.02) GB/s	Two of the three messages go half-way across the ring. The good bandwidth indicates that at least one of these two messages went counterclockwise in (a) and at least one went clockwise in (b). This is used to show that one of the possibilities in #10 cannot occur.

Table 1: *Performance for different communication steps, with 64 KB messages*

5 Affinities and Their Evaluation

Affinities considered

We used the following affinities, other than the default. They were designed to avoid the bottlenecks mentioned in § 4.

- *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 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 this 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.

Communication patterns

We considered the communication patterns specified in table 2. We also considered a few communication patterns that have multiple phases. In each phase, they use one of the simpler patterns mentioned in table 2. They synchronize with a barrier after all the phases are complete. These patterns typically arise in the following collective communication calls: (i) binomial-tree based broadcast, (ii) binomial-tree based scatter, (iii) Bruck algorithm for all-gather, and (iv) recursive doubling based all-gather.

Experimental Results

Figures 5, 6, and 7 show the performance of the different affinity schemes with the communication patterns mention in table 2.

Figure 5 (left) 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

Pattern	Example application
1. 0 ← 7, 1 ← 0, 2 ← 1, ..., 7 ← 6	Ring algorithm, phase 1 of Bruck's algorithm.
2. 0 ↔ 1, 2 ↔ 3, 4 ↔ 5, 6 ↔ 7	Phase 1 of recursive doubling.
3. 0 ↔ 2, 1 ↔ 3, 4 ↔ 6, 5 ↔ 7	Phase 2 of recursive doubling.
4. 0 ↔ 4, 1 ↔ 5, 2 ↔ 6, 3 ↔ 7	Phase 3 of recursive doubling.
5. 0 ← 2, 1 ← 3, 2 ← 4, 3 ← 5 4 ← 6, 5 ← 7, 6 ← 0, 7 ← 1,	Phase 2 of Bruck – allgather.
6. 1 ← 0, 3 ← 2, 5 ← 4, 7 ← 6	Phase 3 of binomial-tree.

Table 2: *Communication patterns studied. The numbers refer to thread ranks.*

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 5 (right) 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.

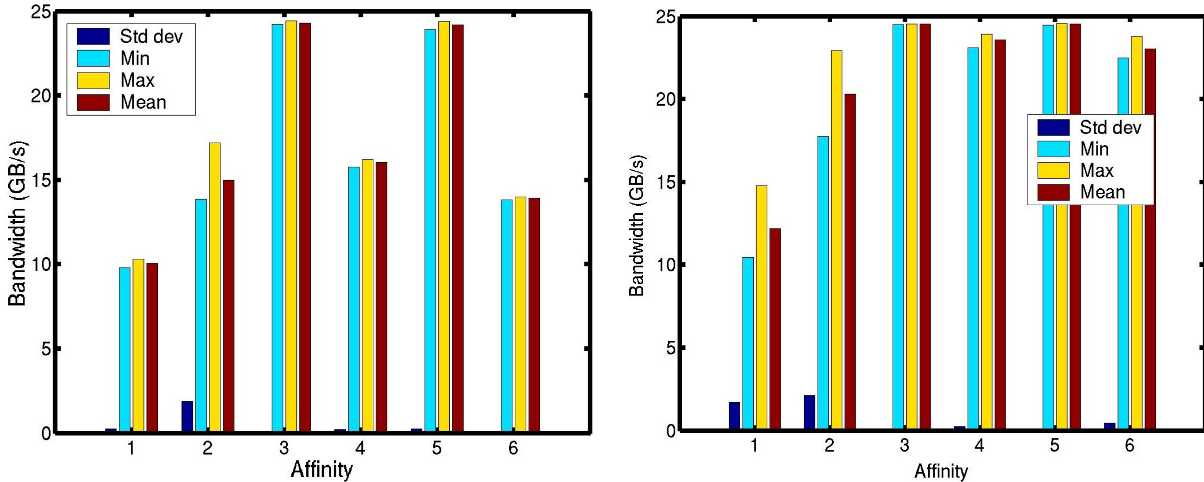

Figure 5: *Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring. Left: Ring pattern. Right: First phase of recursive doubling.*

Figure 6 (left) 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 deterministic patterns do not have any overlap in the same direction. The third phase of recursive doubling is shown in fig. 6 (right). 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.

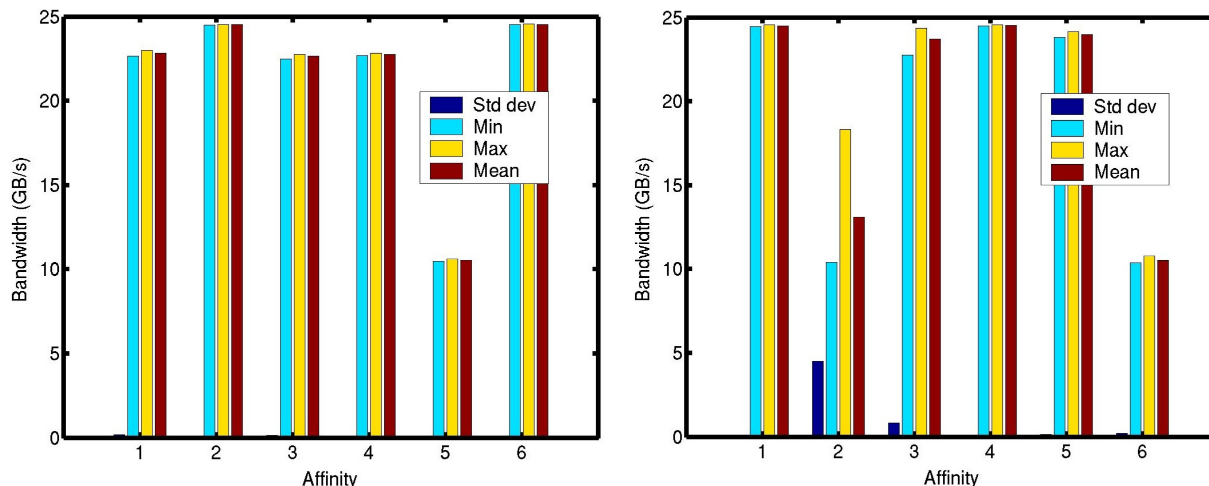

Figure 6: *Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring. Left: Second phase of recursive doubling. Right: Third phase of recursive doubling.*

Figure 7 (left) shows results for the second phase of the Bruck algorithm[4]. 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[5], shown in fig. 7 (right). 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 results on communication patterns with multiple phases. Additional synchronization is involved in these calls, and so the time taken is greater than the sum of the simpler steps. Some of these operations have varying message sizes in each phase. We evaluated them with message sizes such that the largest message is 64 KB, except for

[4]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.

[5]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.

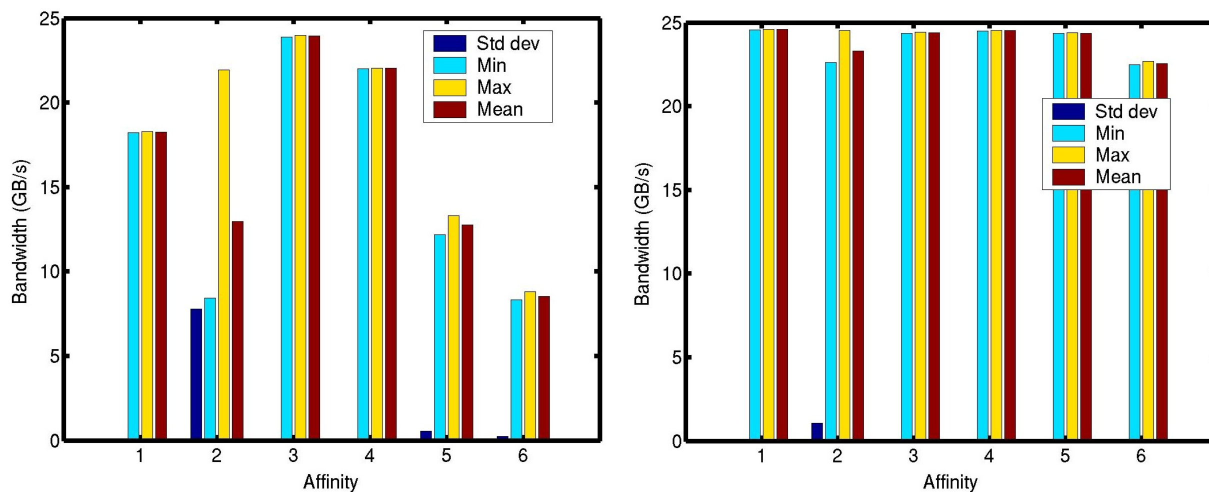

Figure 7: *Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring. Left: Second phase of Bruck algorithm for all-gather. Right: Third phase of binomial-tree.*

the binomial-tree broadcast pattern, which used messages of size 128 KB. In the binomial-tree based patterns, all the affinities were around equally good, taking around 17 μs for the broadcast pattern, and around 5 μs for the scatter pattern. The theoretical limits are 15μs and 4.375μs respectively, based on the SPE receive bandwidth limit of 25.6 GB/s. The synchronizations take additional time. The performance of all the affinities can be explained by their good performance on the third phase of the binomial-tree algorithm. The other two phases of this algorithm do not cause contention, and so yield good performance on all affinities. We do not present figures for the broadcast and scatter patterns, because the performances for all affinities are similar.

Figure 8 shows the results with Bruck[6] and recursive doubling patterns, which occur in all-gather. The EvenOdd affinity performs best with Bruck's pattern. This is not surprising, because this affinity performs well in each phase of Bruck's algorithm (ring, Bruck phase 2, and recursive doubling phase 3). The Identity and Leap affinities perform poorly in one phase each, and so their performance is a little lower than EvenOdd. The Overlap affinity performs poorly in the first phase, and a little lower in the third phase, and so has a lower performance than the previous two affinities. The Ring affinity performs poorly in two phases, and so has the worst performance. EvenOdd and Identity perform best with the recursive doubling pattern, because they do well in each of its phases. The other three deterministic algorithms perform poorly on one phase each, and so yield worse performance.

In all the above results, the EvenOdd affinity performs well. We also performed a few experiments with random permutations of the thread number from which each SPE receives. In these, the EvenOdd affinity was the best, except for one instance in which one particular affinity generated by default was around 5 % faster. Therefore, the EvenOdd permutation appears to be a good one to use, in general. Of course, one can develop a specific communication pattern for which this affinity performs poorly. When one has some specific target

[6]This algorithm can also be used for a barrier; but the small message sizes in the barrier do not cause contention.

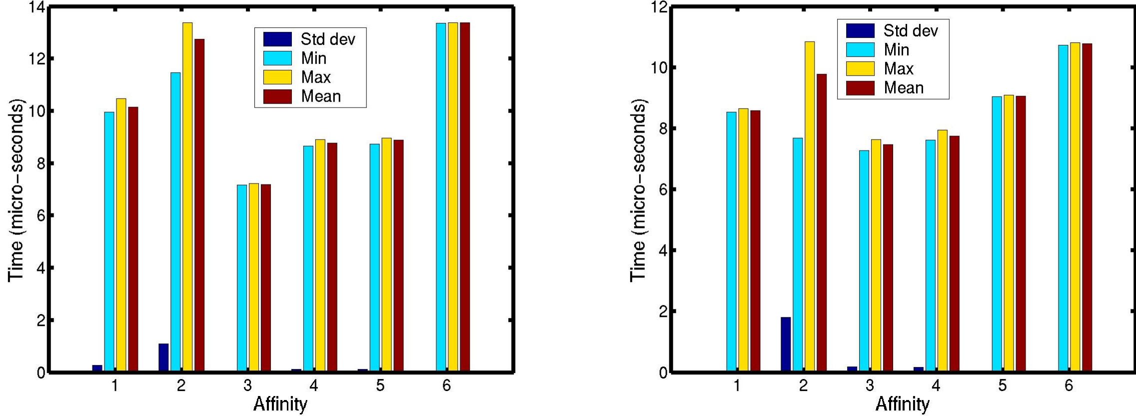

Figure 8: *Performance of the following affinities: (1) Overlap, (2) Default, (3) EvenOdd, (4) Identity, (5) Leap2, (6) Ring, when used in two all-gather patterns. Left: Bruck pattern. Right: Recursive doubling pattern.*

application, one can develop a customized affinity for its communication pattern, using the rules of the thumb mentioned in § 4.

6 Related Work

A detailed presentation of the Cell communication network is given in [2]. They also show results where some operations with contention (including phase 1 of recursive doubling, which is referred to there as *pairwise traffic*) yield sub-optimal bandwidth. However, they have not considered changing affinities to improve performance.

A detailed analysis of collective communication on the Cell is presented in [6]. However, the collective communication in that paper assumes that data is in main memory, and deals with transfer of data between main memory locations, rather than between local stores. That paper too does not consider the effect of affinity.

There has been some work done on the influence of affinity, especially on SMPs [1, 3, 4, 5]. The goal in these papers is to optimize affinity to make more effective use of cache (or local memory). In contrast, our study is based on the features of the on-chip communication network.

7 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. Specifying a good affinity can improve performance by a factor of two over using the default assignment in many cases. Furthermore, the performance is more predictable. We have also shown that the EvenOdd affinity yields good performance on all the patterns studied. If an application has a communication pattern different from any of those presented here, then we have given a few rules of the thumb that can enable application developers to develop a good affinity.

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. This 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.

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. Most of all, we thank Sri S.S. Baba, whose inspiration and help were crucial to the success of this work.

References

[1] A. Foong, J. Fung, and D. Newell. An in-depth analysis of the impact of processor affinity on network performance. In *Proceedings of the IEEE International Conference on Networks (ICON)*, pages 244–250, 2004.

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

[3] E.P. Markatos and T.J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5:379–400, 1994.

[4] J.D. Salehi, J.F. Kurose, and D. Towsley. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing (extended version). *IEEE/ACM Transactions on Networking*, 4:516–530, 1996.

[5] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-CMT multiprocessors. *ACM SIGOPS Operating Systems Review*, 41:47–58, 2007.

[6] 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.