# Characterizing the Basic Synchronization and Communication Operations in Dual Cell-Based Blades[*]

José L. Abellán, Juan Fernández, and Manuel E. Acacio

Dept. de Ingeniería y Tecnología de Computadores, Spain
{jl.abellan,juanf,meacacio}@ditec.um.es

**Abstract.** The Cell Broadband Engine (Cell BE) is a heterogeneous chip-multiprocessor (CMP) architecture to offer very high performance, especially on game and multimedia applications. The singularity of its architecture, nine cores of two different types, along with the variety of synchronization and communication primitives offered to programmers, make the task of developing efficient applications very challenging. This situation gets even worse when we consider Dual Cell-Based Blade architectures where two separate Cells can be linked together through a dedicated high-speed interface. In this work, we present a characterization of the main synchronization and communication primitives provided by dual Cell-based blades under varying workloads. In particular, we focus on the DMA transfer mechanism, the mailboxes, the signals, the read-modify-write atomic operations, and the time taken by thread creation. Our performance results expose the bottlenecks and asymmetries of these platforms which must be taken into account by programmers for improving the efficiency of their applications.

## 1 Introduction

Nowadays, among all contemporary CMP (or chip-multiprocessor) architectures, there is one that is currently concentrating an enormous attention due to its architectural particularities and tremendous potential in terms of sustained performance: the Cell Broadband Engine (Cell BE from now on). From the architectural point of view, the Cell BE can be classified as a heterogeneous CMP. In particular, the first generation of the chip integrates up to nine cores of two distinct types [1]. One of the cores, known as the *Power Processor Element* or PPE, is a 64-bit multithreaded Power-Architecture-compliant processor with two levels of on-chip cache that includes the vector multimedia extension (VMX) instructions. The main role of the PPE is to coordinate and supervise the tasks performed by the rest of cores. The remaining cores (a maximum of 8) are called *Synergistic Processing Elements* or SPEs and provide the main computing power of the Cell BE.

[*] This work has been jointly supported by the Spanish MEC and European Comission FEDER funds under grants “Consolider Ingenio-2010 CSD2006-00046” and “TIN2006-15516-C04-03”.

The Cell BE provides programmers with a broad variety of communication and synchronization primitives between the threads that comprise parallel applications, which were evaluated in [2]. At the end, the performance achieved by the applications running on the Cell BE will depend in great extent on the ability of the programmer to select the most adequate primitives as well as their corresponding configuration values. The main purpose of this work is to expose the performance bottlenecks and asymmetries of those primitives under varying workloads on a dual Cell-based blade.

The rest of the paper is organized as follows. In Section 2 we provide a short revision of the architecture of the Cell BE and a dual Cell-based blade, and a description of some of the communication and synchronization primitives provided to programmers. Next, in Section 3 we introduce our tool, which is called CellStats, for characterizing these primitives. The results obtained after executing CellStats on a dual Cell-based blade are presented in Section 4. Finally, Section 5 gives the main conclusions of the paper and some of the lessons learned that can help programmers to identify the most appropriate primitive in different situations.

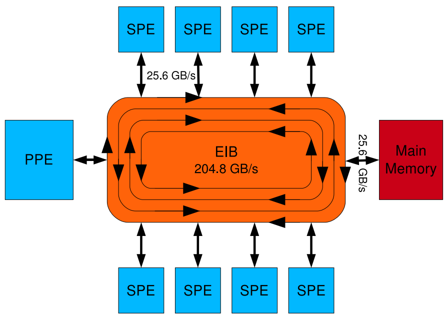
## 2 Dual Cell-Based Blade

### 2.1 Architecture

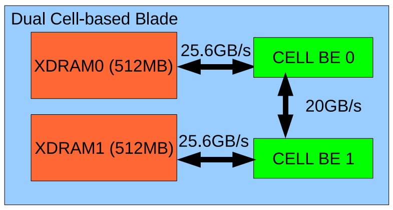
The Cell BE architecture [1] is a heterogeneous multi-core chip composed of one general-purpose processor, called *PowerPC Processor Element* (PPE), eight specialized co-processors, called *Synergistic Processing Elements* (SPEs), a high-speed memory interface controller, and an I/O interface, all integrated in a single chip. All these elements communicate through an internal high-speed *Element Interconnect Bus* (EIB) (see Figure 1(a)).

Each SPE is a 128-bit RISC processor designed for high-performance on streaming and data-intensive applications [3]. Each SPE consists of a *Synergistic Processing Unit* (SPU) and a *Memory Flow Controller* (MFC). The SPUs are in-order processors with two pipelines and 128 128-bit registers. All SPU instructions are inherently SIMD operations that the proper pipeline can run at four different granularities. As opposed to the PPE, the SPEs do not have a private cache memory. In contrast, each SPU includes a 256 KB LS memory to hold both instructions and data of SPU programs, that is, the SPUs cannot access main memory directly. The MFC contains a *DMA Controller* and a set of memory-mapped registers called *MMIO Registers*. Each SPU can write its MMIO registers though several *Channel Commands*. The DMA controller supports DMA transfers among the LSs and main memory. These operations can be issued by the owner SPE, which accesses the MFC through the channel commands, or the other SPEs (or even the PPE), which access the MFC through the MMIO registers.

A dual Cell-based Blade is composed of two separate Cell BEs linked together through the EIB, therefore the maximum theoretical performance is duplicated with respect to that of one Cell BE, which is very interesting for emerging



(a) Block Diagram of Cell BE.



(b) Block Diagram of a Dual Cell-Based Blade.

**Fig. 1.** Cell BE Architecture

scientific, game and multimedia applications. The main components of a dual Cell-based blade are shown in Figure 1(b). In this architecture the two Cell BEs operate in SMP mode with full cache and memory coherency. Main memory is split into two different modules, namely XDRAM0 and XDRAM1, that are attached to Cell0 and Cell1 respectively. In turn, the EIB is extended transparently across a high-speed coherent interface running at 20 GBytes/second in each direction.

### 2.2 Programming

The SPEs use DMA transfers to read from (Get) or write to (Put) main memory. DMA transfer size must be 1, 2, 4, 8 or a multiple of 16 Bytes up to a maximum of 16 KB. DMA transfers can be either blocking or non-blocking. The latter allow overlapping computation and communication: there might be up to 128 simultaneous transfers between the eight SPE LSs and main memory. In addition, an SPE can issue a single command to perform a list of up to 2048 DMA transfers, each one up to 16 KB in size. In all cases, peak performance can be achieved when both the source and destination addresses are 128-Byte aligned and the size of the transfer is an even multiple of 128 Bytes [4]. Mailboxes are FIFO queues that support exchange of 32-bit messages among the SPEs and the PPE. Each SPE includes two outbound mailboxes, called *SPU Write Outbound Mailbox* and *SPU Write Outbound Interrupt Mailbox*, to send messages from the SPE; and a 4-entry inbound mailbox, called *SPU Read Inbound Mailbox*, to receive messages. Every mailbox is assigned a channel command and a MMIO register. The former allows the owner SPE to access the outbound mailboxes. The latter enables remote SPEs and the PPE to access the inbound mailbox. In contrast, signals were designed with the only purpose of sending notifications to the SPEs. Each SPE has two 32-bit signal registers to collect incoming notifications. A signal register is assigned a MMIO register to enable remote SPEs and the PPE to send individual signals (*overwrite mode*) or combined signals (*OR mode*) to the owner SPE. Read-modify-write atomic operations enable simple transactions on single words residing in main memory. For example, the *atomic_add_return* atomic operation adds a 32-bit integer to a word in main memory and returns its value before the addition.

Programming of a dual Cell-based blade is equivalent to that of an independent Cell from a functional point of view. However, there are two important differences. First, dual Cell-based blades have 16 SPEs at programmer's disposal rather than 8 SPEs. This feature involves doubling the maximum theoretical performance but also making much more difficult to extract thread-level parallelism from applications. Second, from an architectural point of view, any operation crossing the Cell-to-Cell interface results in significantly less performance than those that stay on-chip (see Section 4). These facts must be taken into account by programmers to avoid unexpected and undesirable surprises when parallelizing applications for a dual Cell-based blade platform.

## 3 CellStats

### 3.1 Architecture

CellStats is a command-line tool which admits a number of parameters such as the operation to evaluate, the number of SPEs, the specific Cell or Cells to use, the number of iterations and other operation-specific parameters. However, the process to launch, instruct and synchronize the threads is the same in all cases. First, the PPE marshals an structure called control block. The control block contains all the information needed by each SPE to complete the operation demanded by the user. Next, the PPE creates as many threads as specified by the user and synchronizes them using mailboxes. In turn, SPEs transfer the control block from main memory to their private LSs, report control block transfer completion to the PPE, and wait for PPE's approval to resume execution. Then, each SPE performs the task entrusted by the user in a loop. In order to measure the time to complete the loop, the SPE utilizes a register called *SPU_Decrementer* which decrements at regular intervals or *ticks*[1]. Upon completion of the loop, the SPE sends to the PPE the number of elapsed ticks through its outgoing mailbox. In this way, the PPE can compute not only the elapsed time from the go-ahead indication given to the SPEs, but also the time taken by each individual SPE to complete the task. For further details refer to [2].

### 3.2 Functionality

CellStats performs a different experiment depending on the parameters specified by the user: thread creation; PPE-to-SPE or SPE-to-SPE synchronization using mailboxes or signals; data transfers from main memory to local LS/local LS to main memory or remote LS to local LS/local LS to remote LS through DMA operations or list of DMA operations; and atomic operations such as `fetch&add`,

[1] Duration of every *tick* for the dual Cell-based blade is 70 ns.

`fetch&sub`, `fetch&inc`, `fetch&dec`, and `fetch&set` on main memory locations. Besides, it is possible to specify the XDRAM memory module (0 or 1) in which memory buffers are allocated. CellStats manages position of memory buffers by using the `numactl` command.

Thread creation. This operation measures the time to launch the threads that are executed by the SPEs. To do that, an empty task that returns immediately is used. Consequently, this operation takes into account not only the time to create the threads but also the time needed to detect their finalization.

Mailboxes. This operation performs a PPE-to-SPE or an SPE-to-SPE synchronization using mailboxes. The PPE/SPE writes a message in the incoming mailbox (*SPU Read Inbound Mailbox*) of the receiver SPE. Next, the receiver SPE reads the message and replies with another message written to its outgoing mailbox (*SPU Write Outbound Mailbox*). When the initiator SPE/PPE reads the message, the synchronization process is complete. In the former case, the PPE uses the runtime management library function `spe_write_in_mbox` [5] involving a system call which explains the increased latency. Nevertheless, the PPE can also write directly into the corresponding SPE's MMIO register using a regular assignment.

Signals. Unlike mailboxes, this operation performs a PPE-to-SPE or an SPE-to-SPE synchronization using signals. The initiator SPE/PPE signals the destination SPE by writing to the corresponding MMIO register (*SPU Signal Notification*). If the initiator is an SPE, the destination SPE signals in turn the source SPE, thus finishing the synchronization cycle. Otherwise, the destination SPE sends the reply to the PPE using its outgoing mailbox (*SPU Write Outbound Mailbox*). Like mailboxes, it is possible to write directly into the SPE's MMIO register instead of using the runtime management library function call `spe_write_signal` [5].

Atomic operations. These operations enable sequences of read-modify-write instructions on main memory locations in an atomic fashion performed by as many SPEs as indicated by the user. The memory location accessed by the SPEs can be shared or private. In the latter case, the user can also specify the distance, measured in Bytes, between two consecutive private variables.

DMA operations. Data transfers between main memory and the local LS, or between a remote LS and the local LS, are achieved through DMA operations. The user can specify not only the DMA size but also whether the source buffer (GETS) or the destination buffer (PUTS) is shared or private, and whether the memory location is in main memory or in an SPE's LS. Just like atomic operations, the user can specify the distance, measured in Bytes, between two consecutive private buffers.

## 4 Evaluation

### 4.1 Testbed

To develop CellStats we used the IBM SDK v2.1 for the Cell BE architecture installed atop Fedora Core 6 on a regular PC [6]. This development kit includes a simulator, named Mambo, that allows programmers to execute binary files compiled for the Cell BE architecture. To obtain the experimental results, we installed the same development kit atop Fedora Core 6 on a dual Cell-based IBM BladeCenter QS20 blade which incorporates two 3.2 GHz Cell BEs v5.1, namely Cell0 and Cell1, with 1 GByte of main memory and a 40 GB hard disk.

## 4.2 Results

Thread creation. The average latency for launching each new thread, as described in Section 3.2, is considerably high, around 1.68 ms. In order to reduce the cost introduced by thread management, programmers can create SPE threads at startup and keep them alive until the application finishes. In this way, the PPE can submit tasks to the SPE threads by means of communication primitives such as mailboxes or signals, thus minimizing overhead.

Mailboxes and Signals. In Table 1, the average latencies, measured in nanoseconds, for PPE-to-SPE synchronization using mailboxes or signals are shown. In both cases, the PPE can either invoke a system call (Mailbox-sc or Signal-sc) or write directly into the corresponding SPE's MMIO register (Mailbox or Signal). Besides, we consider that the selected SPE can be placed on either Cell for comparison (PPE-SPEc0 for Cell0 and PPE-SPEc1 for Cell1). As we can see, the latency is shorter when writing directly into the SPE's MMIO registers, as defined in file `cbe_mfc.h`, instead of using the runtime management library function calls `spe_write_signal` or `spe_write_in_mbox` [5]. In the former case, it is worth noting that the synchronization latency doubles when the destination SPE resides on Cell1 in both cases. In addition, Table 1 summarizes the average SPE-to-SPE synchronization latency, measured in nanoseconds, using mailboxes or signals when both SPEs are located on the same Cell (SPEc0-SPEc0) or on different Cells (SPEc0-SPEc1), respectively. In the former case, the latency is almost four times shorter because the synchronization messages stay on-chip and do not need to cross the Cell-to-Cell interface.

**Table 1.** Average latency for PPE-to-SPE and SPE-to-SPE synchronization

| Primitive | PPE-SPEc0 | PPE-SPEc1 | SPEc0-SPEc0 | SPEc0-SPEc1 |
|---|---|---|---|---|
| Mailbox-sc | 10,000.0 | 10,000.0 | N/A | N/A |
| Mailbox | 779.7 | 1678.2 | 158.1 | 589.9 |
| Signal-sc | 18,000.0 | 18,000.0 | N/A | N/A |
| Signal | 503.8 | 1182.3 | 160.1 | 619.4 |

Atomic Operations. The average latency of the `fetch&add` atomic operation for a single variable is shown in Figure 2. By using `numactl`, we have selected the variable's memory location (XDRAM0 or XDRAM1). As we can see, latency remains constant, at approximately 111 ns, when the variable is privately accessed by the SPEs. However latency grows linearly, up to 7.5 $\mu$s for 16 SPEs, when the

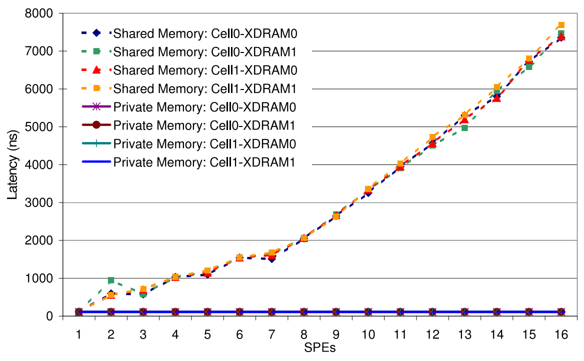


**Fig. 2.** Latency of `fetch&add` on shared and separate variables (128-Bytes stride)

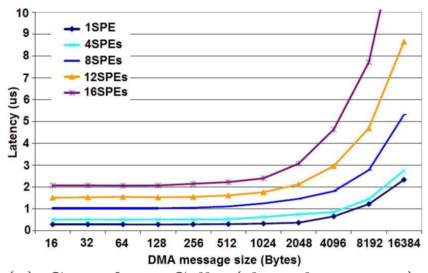
variable is shared by all intervening SPEs. This is due to the fact that shared variables serialize the execution of atomic operations. Results for the rest of the atomic operations are similar and, therefore, have been omitted for the sake of brevity. Notice that the XDRAM memory module employed has negligible effect on performance results. This is because of the small size of the variable (4 Bytes).

DMA Operations. There are three different scenarios for data movement: data transfers between main memory and an SPE's LS (Gets), data transfers between an SPE's LS and main memory (Puts) and data transfers between SPEs' LSs (Movs). Results for Puts do not report significant differences to those of Gets and, therefore, have been omitted for the sake of brevity.

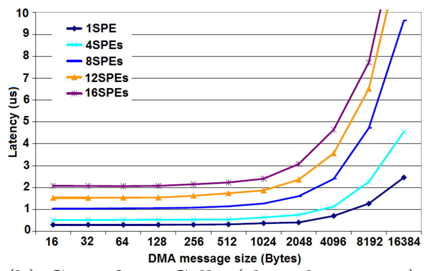
In Figure 3 latency and bandwidth figures for Gets using Cell0 and Cell1 are shown. In particular, to generate Figures 3(a), 3(c), 3(e) and 3(g) (left side) all SPEs from Cell0 were used before any SPEs from Cell1, while to generate Figures 3(b), 3(d), 3(f) and 3(h) (right side) SPEs were used in the opposite order. As we can see, two general trends can be identified. First, latency is constant for message sizes smaller than or equal to the cache line, that is 128 Bytes. Second, latency grows proportionally to the message size for messages larger than the cache line until the available bandwidth is exhausted. In addition, a more in depth analysis provides other interesting conclusions. Latency is constant, but proportional to the number of SPEs, for message sizes up 128 Bytes regardless of the originating Cell when shared buffers are used (see Figures 3(a) and 3(b)). Latency is constant, around 300 ns, for message sizes up 128 Bytes regardless of the originating Cell when private buffers are used (see Figures 3(c) and 3(d)).[2]

For bandwidth figures, there are three important trends to be considered. Firstly, when 8 SPEs are involved, Gets initiated in Cell0 obtain an aggregate bandwidth of 24.6 GB/s (close to the peak memory bandwidth), while Gets initiated in Cell1 reach an aggregate bandwidth of 13.6 GB/s. This is due to the fact that buffers are always placed in XDRAM0 memory module. Therefore, Gets from SPEs in Cell1 must cross the Cell-to-Cell interface, limiting the

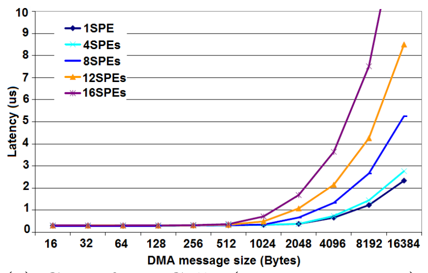
[2] Stride is larger than or equal to the cache line size in all cases.



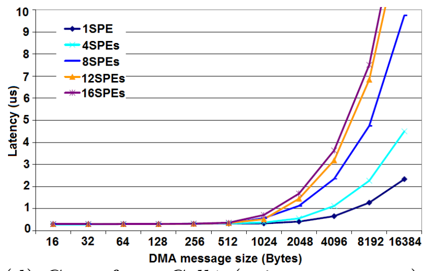
(a) GETs from Cell0 (shared memory)



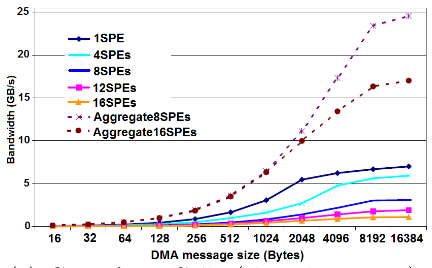
(b) GETs from Cell1 (shared memory)



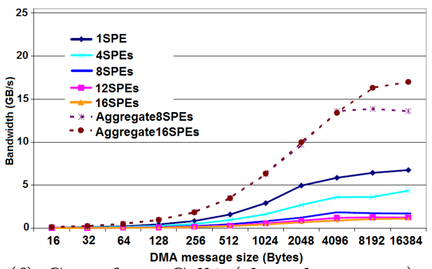
(c) GETs from Cell0 (private memory)



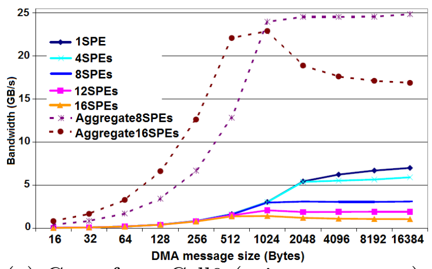
(d) GETs from Cell1 (private memory)



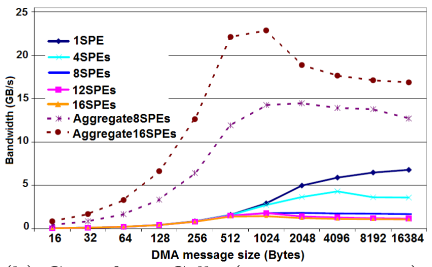
(e) GETs from Cell0 (shared memory)



(f) GETs from Cell1 (shared memory)



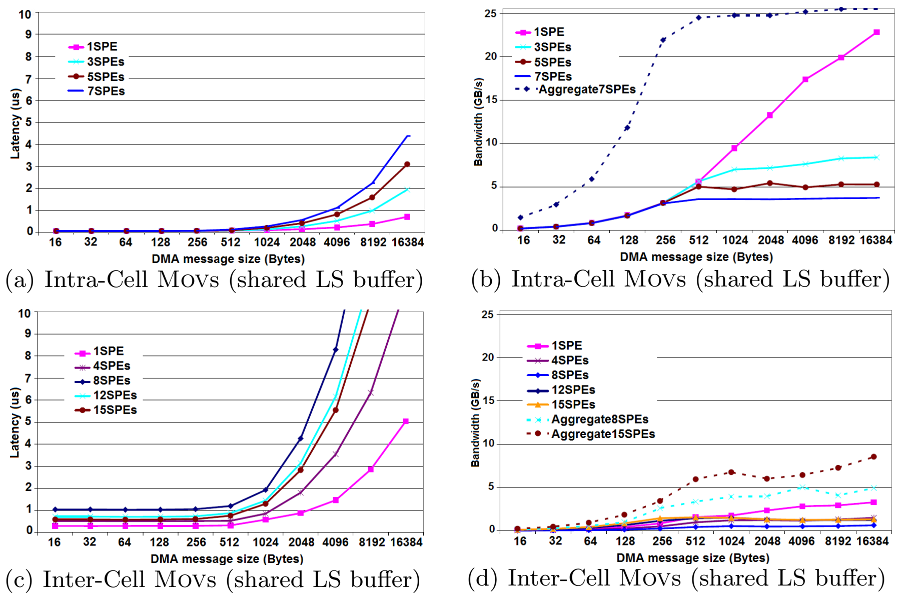
(g) GETs from Cell0 (private memory)



(h) GETs from Cell1 (private memory)

**Fig. 3.** Latency and bandwidth of DMA GETs on shared and private main memory buffers for a variable number of SPEs and packet sizes using Cell0 and Cell1

maximum achievable aggregate bandwidth. With the `numactl` command, we have verified that allocating all buffers in XDRAM1 memory module reports just the opposite results. Secondly, when 16 SPEs are considered both Cells are



(a) Intra-Cell Movs (shared LS buffer) (b) Intra-Cell Movs (shared LS buffer)

(c) Inter-Cell Movs (shared LS buffer) (d) Inter-Cell Movs (shared LS buffer)

**Fig. 4.** Latency and bandwidth of Movs on shared LS buffers for a variable number of SPEs and packet sizes using a single Cell and both Cells

involved, thus the figures report the benefits of transferring data from the closest XDRAM memory module (for SPEs within Cell0), and also report the drawback of going through the Cell-to-Cell interface (for SPEs within Cell1). Finally, for private buffers the aggregate bandwidth grows faster for message sizes up to 1 KB because of exploiting simultaneous transfers to different buffers. After that, the aggregate bandwidth figures converge to the same values as before. In turn, latency and bandwidth figures for Movs using Cell0 and Cell1 are shown in Figure 4. In particular, Figures 4(a) and 4(b) correspond to DMA Movs in Cell0 , while Figures 4(c) and 4(d) correspond to DMA Movs between Cell0 and Cell1. In the former case, SPEs approach the maximum available bandwidth of the EIB-to-SPE interface. In the later case, the Cell-to-Cell interface bandwidth is the limiting factor. Nevertheless, the latency is much longer than expected resulting in an aggregate bandwidth shorter than that of Gets originating in Cell1.

## 5 Conclusions

In this work, we have evaluated the synchronization and communication mechanisms of the Cell BE on a dual Cell-based blade platform. In this way, we can give some recommendations for dual Cell-based blade programmers such as: programmers should avoid frequent creation of threads, since thread creation introduces a significant overhead; they should use direct writes to the SPEs'

MMIO registers, since using runtime management library calls is very slow; for atomic operations, whenever possible, they should use private buffers residing on different cache memory lines, because latency of shared buffers grows linearly with the number of involved SPEs; in case of DMA transfers, they should use private buffers up to 1KB. For messages larger than 1KB, the latency is identical in both cases; finally programmers should be aware of the Cell-to-Cell interface, which determines the maximum achievable bandwidth, and also the asymmetries that arise when memory locations are in the furthest XDRAM memory module. This can be controlled by using the `numactl` command.

## References

1. Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., Shippy, D.: Introduction to the Cell Multiprocessor. IBM Journal of Research and Development 49(4/5), 589–604 (2005)
2. Abellán, J.L., Fernández, J., Acacio, M.E.: CellStats: a Tool to Evaluate the Basics Synchronization and Communication Operations of the Cell BE. In: Proceedings of $16^{th}$ Euromicro International Conference on Parallel Distributed and network-based Processing, pp. 261–268 (2008)
3. Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic Processing in Cell's Multicore Architecture. IEEE Micro 26(2), 10–24 (2006)
4. Kistler, M., Perrone, M., Petrini, F.: Cell Processor Interconnection Network: Built for Speed. IEEE Micro 25(3), 2–15 (2006)
5. IBM Systems and Technology Group: SPE Runtime Management Library Version 2.1. (2007)
6. IBM Systems and Technology Group: Cell Broadband Engine Software Development Toolkit (SDK) Installation Guide Version 2.1. (2007)