Performance Impact of Task Mapping on the Cell BE Multicore Processor

Jörg Keller[1] and Ana L. Varbanescu[2]

[1] University of Hagen, 58084 Hagen, Germany
joerg.keller@fernuni-hagen.de
[2] Delft University of Technology, Delft, The Netherlands
a.l.varbanescu@tudelft.nl

Abstract. Current multicores present themselves as symmetric to programmers with a bus as communication medium, but are known to be non-symmetric because their interconnect is more complex than a bus. We report on our experiments to map a simple application with communication in a ring to SPEs of a Cell BE processor such that performance is optimized. We find that low-level tricks for static mapping do not necessarily achieve optimal performance. Furthermore, we ran exhaustive mapping experiments, and we observed that (1) performance variations can be significant between consecutive runs, and (2) performance forecasts based on intuitive interconnect behavior models are far from accurate even for a simple communication pattern.

1 Introduction

Current multicore processors such as the Cell BE or Intel/AMD quad cores appear to the application programmer as symmetric multiprocessors, where the cores are identical (we exclude the PPE core on Cell BE here), and where the interconnect used by the cores to communicate with each other and with the global (off-chip) memory is symmetric, i.e. it appears as if it were a bus with huge bandwidth. Thus, the application programmer must still think about which (micro)tasks of his parallel program are aggregated into one task and mapped onto one core, but he does not have to take care onto which core they are mapped.

There have already been studies concluding that, as the interconnect in reality is more complex than a bus, the concrete mapping does play a role. For example, IBMs own experiments [5] indicate that if Cell SPEs communicate in pairs, then there is an optimal and a worst case mapping, with communication performance that differs by a factor of about 2.5. Ainsworth et al. [1] conclude that among other factors that constrain the usage of Cell's Element Interconnect Bus (EIB), such as the control structure, also the concrete communication pattern leads to performance differences of the same order. Sudheer et al. [12] discuss the influence of thread-SPE mapping for large messages. Their conclusions support our intuition that random thread assignments on SPEs can impact both performance and predictability significantly. Still, the focus of their work is on finding the right affinity for a given communication pattern among a small set

of possible mappings, while we are focusing more on the performance trade-offs between all possible affinities. Furthermore, in addition to the analysis of the interconnect behavior for large packets transfers done in [12], we also experiment with small and medium packets, encountered in many real-life applications (e.g., image processing convolution filters or histogram computations), and we provide interesting insights there as well.

Gross et al. [14] report that asymmetries also exist for Intel Xeon quad-cores. When cores intensively communicate with main memory, the performance depends on which core runs the thread with highest bandwidth demand. Yet, so far the numerous literature on mapping applications to multiple cores (see e.g. the mapping optimization in [9] or the extensive related work section in [8]) does not take this asymmetry into account.

Furthermore, it is also not clear how real applications are affected by this. Besides a worst case and a best case mapping, one would need an average case behavior to decide whether the worst case is a theoretical one and in practice, the mapping done by the runtime system provides close-to-optimal performance, or whether the mapping to particular cores really must be taken into account. Also, the application might exhibit multiple communication patterns so it may be worthwhile to consider all possible mappings to finally pick one that gives best performance over all patterns.

We therefore decided to run a set of experiments on a Cell BE processor with a synthetic benchmark application where the tasks communicate in a bi-directional ring. Our findings are that the mapping is of importance already for small packet sizes, and that the application has little chance to find a mapping with best performance, even when actively influencing the mapping.

The remainder of this article is organized as follows. In Sect. 2 we give the necessary details about the Cell BE processor and our synthetic benchmark application. In Sect. 3 we report the results of our experiments. In Sect. 4, we give a conclusion and outlook on future work.

2 Cell BE and Benchmark Application

We first introduce the Cell BE processor in as much detail as necessary, and then present the synthetic benchmark program used to evaluate the impact of the mapping.

2.1 Cell Broadband Engine

If there is any processor to be credited with starting the "multi-core revolution", the Cell Broadband Engine (Cell/B.E.) must be the one. Originally designed by the STI consortium — Sony, Toshiba and IBM — for the Playstation 3 (PS3) game console, Cell/B.E. was launched in early 2001 and quickly became a target platform for a multitude of HPC applications. Still, being a hybrid processor with a very transparent programming model, in which a lot of architecture-related

optimizations require programmer intervention, Cell/B.E. is also the processor that exposed the multi-core programmability gap.

A block diagram of the Cell processor is presented in Figure 1. The processor has nine cores: one Power Processing Element (PPE), acting as a main processor, and eight Synergistic Processing Elements (SPEs), acting as computation-oriented co-processors. In the PS3, only six SPEs are visible under Linux. An additional SPE runs the hypervisor, and the last SPE is switched off, allowing to use Cell chips with one defective SPE for PS3. It is unclear which of the SPEs from Figure 1 are visible to the user.

For the original Cell (the variant from 2001), the theoretical peak performance is 230 single precision GFLOPS [10] (25.6 GFLOPS per each SPE and for the PPE) and 20.8 double precision GFLOPS (1.8 GFLOPS per SPE, 6.4 GFLOPS per PPE). In the latest Cell version, called PowerXCell 8i, the double precision performance has been increased to 102.4 GFLOPS. All cores, the external main memory, and the external I/O are connected by a high-bandwidth Element Interconnection Bus (EIB), which in reality is composed of four unidirectional rings. The maximum data bandwidth of the EIB is 204.8 GB/s.

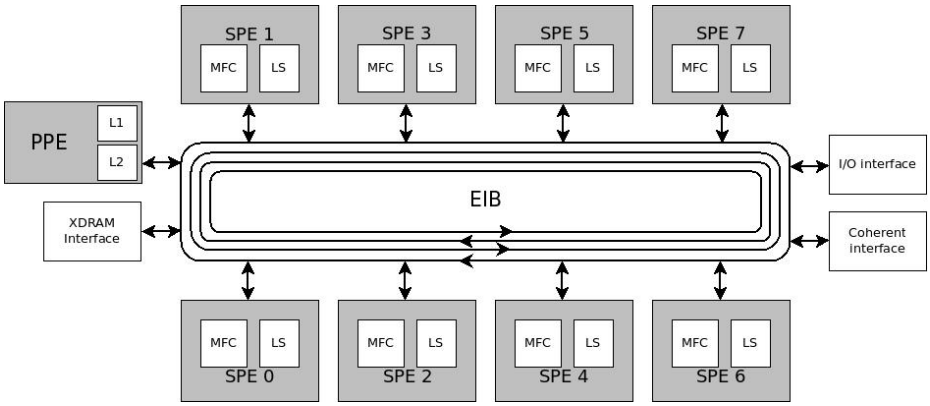

Fig. 1. The Cell Broadband Engine.

The PPE contains the Power Processing Unit (PPU), a 64-bit PowerPC core with a VMX/AltiVec unit, separated L1 caches (32KB for data and 32KB for instructions), and 512KB of L2 Cache. The PPE's main role is to run the operating system and to coordinate the SPEs.

Each SPE contains a RISC-core (the SPU), a 256KB Local Storage (LS), and a Memory Flow Controller (MFC). The LS is used as local memory for both code and data and is managed *entirely* by the application. The MFC contains separate modules for DMA, memory management, bus interfacing, and synchronization with other cores. All SPU instructions are SIMD instructions working on 128-bit operands, to be interpreted e.g. as four 32-bit words. All 128 SPU registers are 128-bit wide. Each SPE has a local address space. Besides that, there is a

global address space that spans the external main memory and the eight local storages. Yet, an SPU can only directly access its own local storage, all other accesses must use explicit DMA transfers to copy data to the local storage.

The Cell/B.E. has been used to speed-up a large spectrum of applications, ranging from scientific kernels [16] to image processing applications [4] and games [6]. The basic Cell/B.E. programming model uses simple multi-threading: the PPE spawns threads that execute asynchronously on SPEs, until interaction and/or synchronization is required. The SPEs can communicate with the PPE using low-level mechanisms like signals and mailboxes for small amounts of data, or using DMA transfers via the main memory for larger data. The SPEs can also transfer data directly between local storages by DMA transfers. In this model, all data distribution, task scheduling, communication, and processing optimizations are performed "manually" by the user (i.e., they are not automated).

Multiple DMA transfers can occur concurrently on each EIB ring, but they must use different parts of that ring. Thus, if SPEs 1 and 3, and 5 and 7 communicate in pairs, this is possible by using only one ring.

Despite its obvious programming complexity, this model allows users to control the access to most of the Cell hardware components explicitly. As we are interested in the performance behavior of the EIB, and its potential influence on a larger application, we are using this model to implement our test application.

2.2 Synthetic Benchmark Application

Our synthetic benchmark application uses $k = 6$ threads on a Playstation 3 with Linux operating system, and $k = 8$ threads on a Cell blade. The main thread on the PPE starts k threads that run on the k SPEs available. Each thread runs a task. The six tasks communicate in a bi-directional ring, i.e. initially, each task creates two packets, then it sends these packets to its neighbor tasks. Then, whenever it receives a packet from a neighbor task, it sends this packet to the other neighbor task on the ring. This is repeated several million times. The packets have a size of 1 kbyte and contain random data. For comparison, we also did runs with packets of size 2 kbyte and 0.5 kbyte.

The application uses a simple, low-overhead message-passing library for Cell BE developed at University of Hagen as a student project [11]. Each transfer is buffered on sender and receiver sides. Upon a send command, a sender transmits a notification to the receiver, the receiver from time to time polls for notifications, and if it has one, initiates a SPE-to-SPE DMA transfer for the packet (the buffer addresses are fixed and communicated beforehand). When the DMA transfer is completed, the receiver sends an acknowledge to the sender. The sender from time to time polls for acknowledges, and if it has one, marks the send buffer as free. The notifications and acknowledgements are communicated via a matrix data structure in global memory. This allows each task to check for all notifications at once by performing one get on a matrix row. Thus, the majority of traffic on the EIB is from SPE-to-SPE data transfer, but there is traffic to and from external memory as well.

The ring communication above is repeated for each mapping of threads to SPEs. When the threads T_i, where $i = 0, \ldots, k-1$ have been started, thread T_i executes task $t_{\pi_0(i)}$. Then the several million ring communications are done, the time needed is recorded, then T_i executes task $t_{\pi_1(i)}$, the ring communications are done, the time is recorded, and so on. On a PS3, we run 120 permutations π_j, where $j = 1, \ldots, 120$ form the $5! = 120$ possible mappings of threads to SPEs with $\pi_j(0) = 0$. We denote a permutation with $\pi(i) = a_i$ for $i = 0, \ldots, k-1$ by $[a_0 \ldots a_{k-1}]$.

Thus, we test all mappings of 6 tasks onto 6 threads, relative to task t_0 which always is executed by thread T_0. This was done to reduce the number of permutations from 720 to 120, so to be able to still manually inspect the performance results with reasonable effort. By doing so, we implicitly test all possible mappings from tasks to SPEs, as the application is symmetric in the sense that all threads are executing identical tasks.

On a Cell blade, we use the $7! = 5,040$ mappings possible with 8 SPEs and $\pi_j(0) = 0$.

As the default mapping of the threads to the SPEs is somewhat random [12], and not necessarilty identical on two runs of the application, we also created a variant of the application where thread T_i is pinned to SPU i. In order not to rely on particular implementations of the library software, we refrained from using `spe_set_affinity`, but used the mapping of the SPEs' local stores into the global address space, and provided task IDs by having the PPE directly write the IDs into a predetermined address at the local stores.

We compiled both variants of our application using IBM Cell SDK 3.1.0.1 with gcc compiler and -O3 option.

3 Experiments

The application used 10^7 transmissions per SPE to get a sufficiently long runtime. We did three runs of the application, averaged the runtime for each permutation, and ranked the permutations according to the average runtime. For this ranking, Fig. 2 depicts the average runtime as well as the runtime curves of the three runs for 1 kbyte packets on the PS3.

First, we notice that the ratio of best to worst average runtime is about 2.35, while it is about 2.8 for the individual runs. This already indicates that there are noticeable differences between the runs for each permutation. Second, the best permutation with respect to average runtime is [014235]. While it is on position 1 in run 3, it is on positions 42 and 45 in runs 1 and 2, respectively. The permutation on rank 2 with respect to average runtime is [025143], which is on positions 11, 9, 8 with respect to the runtime in runs 1, 2, and 3, respectively. Thus, the permutation with rank 2 in any of the runs must perform very badly in the other runs. This is also visible in the graph, where the curves for the individual runs look rather wild, with many peaks downward (i.e. permutations with small runtime) on high ranks in average runtime, corresponding to the curves differing substantially. Third, the average curve is mostly a straight line

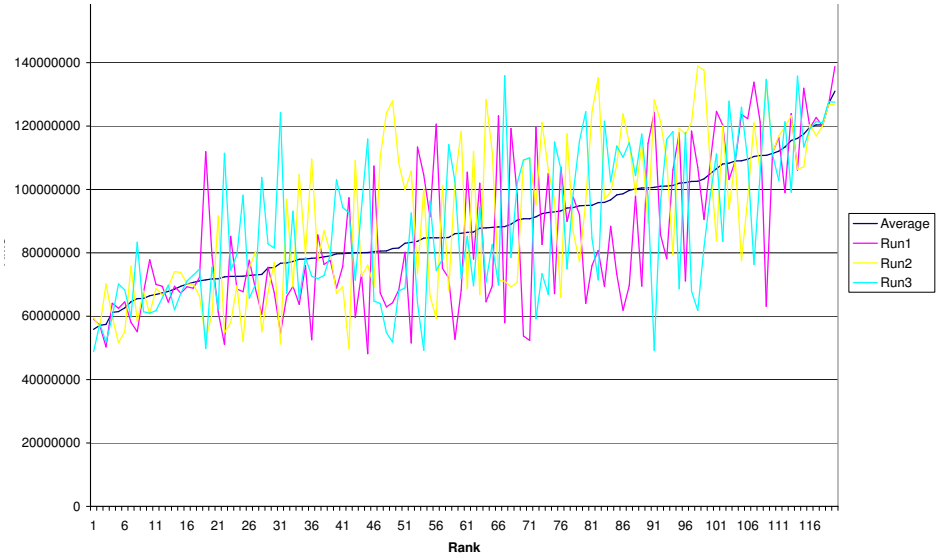

Fig. 2. Runtime differences with default mapping of tasks on PS3

(average and median are very close together), so that one could expect to be a factor of about 1.18 away from the optimal performance when using a random permutation. Fourth, the identical permutation, which one would use by default, is on rank 92 with respect to average runtime, which means that it is by a factor of 1.8 away from the optimum. Finally, the two permutations on the top places do not show any mapping that one would expect, such as [024531] or the like.

We conclude that in this way, no reliable forecast on the performance is possible, and hence a random mapping of tasks to threads might be the best to do.

With the assumption that the mapping of local store addresses into the global address space represents the SPE numbering, we used the variant of the application as described in the previous section. The results for PS3 are depicted in Fig. 3, with the number of rounds increased to $5 \cdot 10^7$ per permutation.

The top 2 permutations [043512] and [013542] with respect to average runtime are on ranks 3,2,4 and 5,7,3 in the individual runs. The individual curves are now closer to the average, the best to worst ratio is about 2.45 for the individual runs and 2.27 for the average. The average curve is still mostly a straight line, but has a rather steep beginning. One of the mappings that would seem to be best, [024531] is on rank 2, the other [013542] is on rank 7, but already has a 25% performance loss. However, the identity permutation has gone down to rank 101 with an average runtime that is longer than the average runtime for the rank 1 permutation by a factor of 1.97. While those mappings (called ring affinity) were considered inferior in [12], that study only used uni-directional

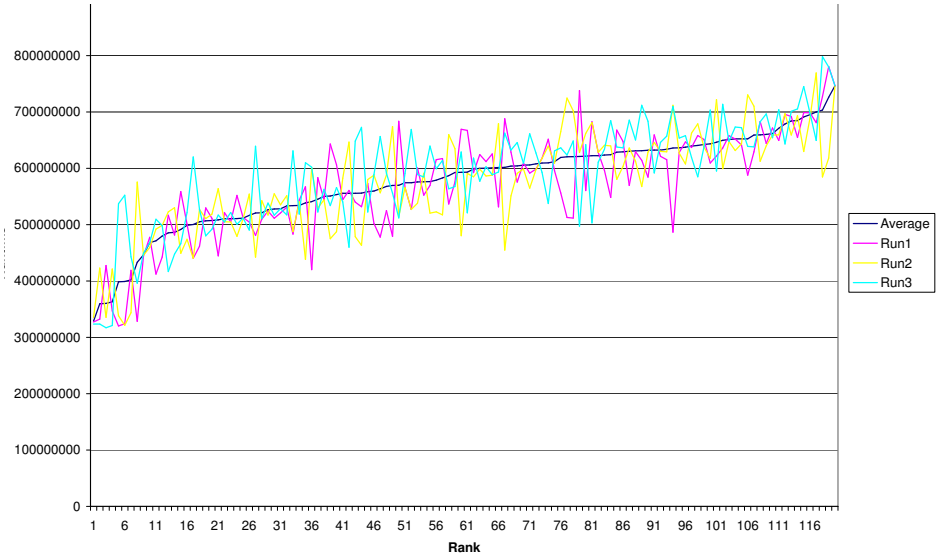

Fig. 3. Runtime differences with deterministic mapping of tasks on PS3

ring communication, while we consider bi-directional ring communication, which seems more realistic.

We repeated the PS3 experiment with packet sizes of 2 kbyte and 0.5 kbyte, but did not see significant changes compared to the previous runs.

We performed the experiment with deterministic mapping also on a QS22 blade of the Cell buzz cluster of Georgia Tech. While each Cell blade contains two Cell processors, we used only one of them. In Fig. 4, the runtimes of the different mappings for different packet sizes are shown. The curves look similar to the curve from Fig. 3. The ratios of worst to best average runtime are 3.4, 5.3, 9.3 for 2, 1, and 0.5 kbyte packets, respectively, indicating that with more traffic (the ring now has 16 links), dependence on the mapping grows. Furthermore, the permutations [02467531] and [01357642], which one would choose intuitively, are not among the top 10. The former is on rank 95 of 5,040, albeit with a runtime increase of 74% compared to rank 1. The latter is only on rank 3,212, with a runtime about 2.6 times the best runtime. We have no explanation of this asymmetry.

We also conclude from the Cell blade experiment that the traffic from the hypervisor SPE in the PS3 setting had no notable influence on the application performance.

For reference, we also restricted our application on the Cell blade to only perform uni-directional ring traffic, see Fig. 5. Here we see that the mapping has a much smaller influence, except for a small number of mappings. This also links our study to the work from [12].

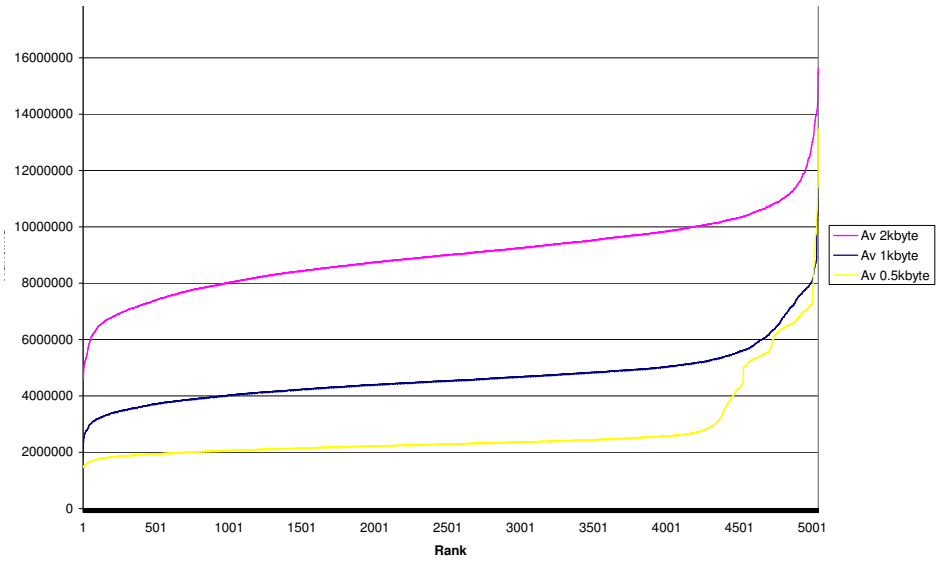

Fig. 4. Runtimes on Cell blade for different packet sizes

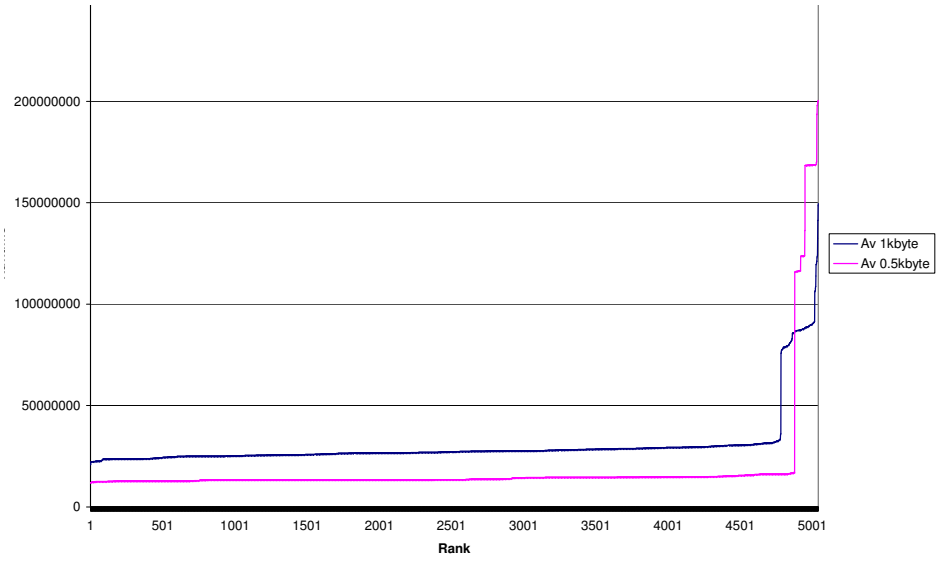

Fig. 5. Runtimes of unidirectional ring on Cell blade with different packet sizes

4 Conclusions and Future Work

We have investigated the influence of the concrete mapping of tasks onto multiprocessor cores on the performance of a parallel application. Our test setting was the Cell BE processor running a set of tasks that communicate in a bidirectional ring. Our findings were that the mapping seems to vary from one run of the application to the next, so that it is impossible to employ a mapping that optimizes performances by exploiting the knowledge about Cell's communication structure. This is even true when trying to pin tasks to cores by writing the task numbers directly to local stores, using the embedding of local store addresses into the global address space, which we assume to be fixed. The performance impact of the mapping is even visible for small messages (1 kbyte), while previous studies indicated an influence only for large messages.

While the Cell BE processor is only one example of a multicore processor with a very particular structure (and with an unclear future), it still can serve as a reference example of the future manycore and the Multiprocessor-System-on-Chip (MPSoC) architectures.

The manycore processors of the future, of which the first are soon to debut, will feature, besides tens of cores, complex on-chip interconnection networks, and local memories for direct core-to-core communication; i.e. they will have a Cell-like structure. A good example is the Intel single-chip cloud (SCC) with 24 tiles in a 6×4 2D-mesh, each tile comprising two IA cores and support for message passing from tile to tile [15]. This support comprises a 16 KB message buffer on each tile (similar to a Cell's local store used for message buffering) and routines for direct tile-to-tile communication. Thus, for these architectures the programmer depends on a communication performance model or a clever runtime system to explicitly or automatically map tasks onto cores such that the resulting communication performance is optimal or close to optimal.

MPSoC architectures, increasingly used in high-end embedded systems, are also moving towards increased complexity, featuring heterogeneous cores, distributed memory, and complex communication networks. This trend, already predicted in 2004 by Wolf [17], is nowadays proven by architectures developed both in the academia, like CoMPSoC [7], and in the industry, like ARM MPCore [2] or ARM Cortex-A9 [3]. Because of either soft or hard real-time requirements, typical for these systems, the interconnection behavior must be known in order for the application to be mapped with communication pattern awareness. Disregarding these issues will lead, much like on the Cell/B.E., on less efficient use of the platform.

In [13], a model to find the optimal mapping of threads to SPEs was proposed for the Cell BE processor, but for ring communication the mapping proposed by the model was inferior to a manually selected mapping. Therefore, while we are not dismissing their results, we argue that interconnect models for performance prediction have to be carefully validated. Also, it seems that using a model to enforce predictable interconnect performance might result in significant, non-intuitive performance penalties.

A number of questions in our case study could not be answered yet. For example, the variance in runtime between different runs in the deterministic mapping case could come from constraints on the EIB control bus, see [1]. Also, our synthetic benchmark is somewhat artificial. It would be interesting to see whether the performance of a larger application, where SPEs communicate frequently in a multitude of patterns that are more complex, will be even more dependent on the concrete mapping. Finally, Cell is a relatively simple test platform as there is almost no interference with the operating system, for the simple reason that there is none on the SPEs. Of course, the operating system on the PPE also uses the bus to communicate with the main memory. On other platforms, things like thread migration may occur and must be taken into account if a predictable, performance-aware mapping is to be achieved. While setting the thread affinity can prevent migration, it creates a tension between application and operating system that somehow should be resolved.

Overall, our study shows that although the memory and core performance issues dominate the multi-core performance studies, their interconnects might as well become a bottleneck for (predictable) performance. Although we are far from an accurate, yet simple analytical model for multi-core interconnect performance, studies such as ours might enable a statistical, platform-specific approach to predictable mappings, which might in turn be used in any predictable and productive parallel programming model for multi-cores.

Acknowledgements

The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research. The authors also would like to thank A. Platonov for running part of the experiments.

References

1. Ainsworth, T.W., Pinkston, T.M.: On characterizing performance of the Cell Broadband Engine Element Interconnect Bus. In: Proc. Int.l Symposium on Networks-on-Chip. pp. 18–29 (May 2007)
2. ARM: ARM11 MPCore processor. `http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php`
3. ARM: Cortex-A9 processor. `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`
4. Benthin, C., Wald, I., Scherbaum, M., Friedrich, H.: Ray tracing on the Cell processor. In: IEEE Symposium of Interactive Ray Tracing. pp. 15–23 (Sep 2006)
5. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation: a performance view. IBM J. Res. Dev. 51(5), 559–572 (2007)
6. D'Amora, B.: Online Game Prototype (white paper). `http://www.research.ibm.com/cell/whitepapers/cellonlinegame.pdf` (May 2005)

7. Hansson, A., Goossens, K., Bekooij, M., Huisken, J.: CoMPSoC: A template for composable and predictable multi-processor system on chips. ACM Trans. Design Autom. Electr. Syst. 14(1) (2009)
8. Keller, J., Kessler, C.W.: Optimized pipelined parallel merge sort on the Cell BE. In: Proc. 2nd Workshop on Highly Parallel Processing on a Chip (HPPC-2008) at Euro-Par 2008. pp. 131–140 (Aug 2008)
9. Kessler, C., Keller, J.: Optimized mapping of pipelined task graphs on the Cell BE. In: Proc. 14th International Workshop on Compilers for Parallel Computers (Jan 2009)
10. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. IEEE Micro 26(3), 10–23 (2006)
11. Platonov, A., Sorokin, S.: Cell programming lab course (in german). `https://ziegel.fernuni-hagen.de/~jkeller/studentproject.pdf` (Mar 2010)
12. Sudheer, C., Nagaraju, T., Baruah, P., Srinivasan, A.: Optimizing assignment of threads to SPEs on the Cell BE processor. In: Proc. 10th Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC at IPDPS 2009) (May 2009)
13. Sudheer, C., Sriram, S., P.K, B.: A communication model for determining optimal affinity on the Cell BE processor. In: Proc. Int.l Conference on High Performance Computing (HiPC 2009) (Dec 2009)
14. Tuduce, I., Majo, Z., Gauch, A., Chen, B., Gross, T.R.: Asymmetries in multi-core systems — or why we need better performance measurement units. In: Proc. Exascale Evaluation and Research Techniques Workshop (EXERT) at ASPLOS 2010 (Mar 2010)
15. van der Wijngaart, R., Mattson, T.: RCCE: a small library for many-core communication. In: Proc. Intel Labs Single-chip Cloud Computer Symposium. `http://techresearch.intel.com/UserFiles/en-us/File/SCC_Sympossium_Mar162010_GML_final.pdf` (Mar 2010)
16. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell processor for scientic computing. In: ACM Computing Frontiers 06. pp. 9–20 (May 2006)
17. Wolf, W.: The future of multiprocessor systems-on-chips. In: Proc. 41st Design Automation Conference (DAC '04). pp. 681–685 (Jul 2004)