Evaluating the Portability of UPC to the Cell Broadband Engine

Ruben Niederhagen, Stefan Lankes

Chair for Operating Systems, RWTH Aachen University,
Templergraben 55, 52056 Aachen, Germany
{ruben,lankes}@lfbs.rwth-aachen.de

Abstract. *Unified Parallel C* (UPC) is a parallel programming language for distributed as well as shared memory systems. The *Cell Broadband Engine* (Cell BE) is a state of the art multicore processor. In this paper we evaluate the opportunities and pitfalls of implementing the Berkeley UPC runtime system API for the Cell BE and thus bringing UPC to Cell. We propose a mapping of the distributed shared memory model of UPC to the memory model of the Cell architecture. To achieve acceptable performance of UPC applications on the Cell processor, the specific properties of this processor need to be addressed. We discuss data caching as an optimization strategy to hide communication latency between main memory and the computations cores.

1 Introduction

Parallel computing has a long history in scientific and high performance computing. A variety of programming standards and languages are in use, e.g., the standards OpenMP and MPI for C, C++, and FORTRAN or parallel programming languages like Co-Array FORTRAN and Unified Parallel C. They are used for different architectures like SMP (Symmetric Multiprocessing, e.g., current multicore processors), NUMA (Non-Uniform Memory Access, e.g., SGI Altix) and NoRMA (No Remote Memory Access, e.g., BlueGene).

The Cell Broadband Engine was released in 2005 by Toshiba, Sony, and IBM. It introduces a new parallel architecture, which has some aspects in common with the established massively parallel systems (it is often referred to as 'cluster on a chip'). A transfer of the programming paradigms of these systems to the Cell processor offers the following advantages: not only would programmers be able to use their well developed knowledge without having to learn new specific mechanisms, but also porting of existing applications to the new hardware would be easier.

This article is structured as follows: UPC and its memory model are introduced in section 2. In section 3 the Cell Broadband Engine will be discussed with focus on the memory architecture. Section 4 puts UPC and the Cell together and presents a mapping of UPC's memory model with the memory model of the Cell processor. In section 5 a software managed cache is presented as an improvement for remote memory accesses. The final part gives a short summary and a prospect to open questions and future tasks.

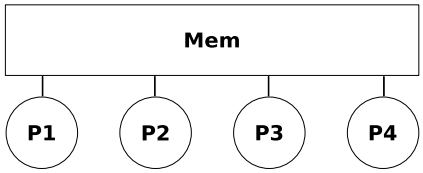

Fig. 1. Shared Memory

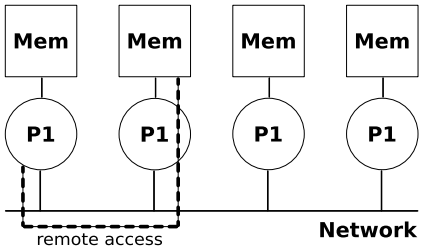

Fig. 2. Distributed Memory

2 UPC Memory Model

Unified Parallel C (UPC) [1, 2] is a C-language derivate for parallel computing. UPC targets parallel architectures with either shared memory or distributed memory. Fig. 1 and Fig. 2 illustrate these memory models. On shared memory systems every computation node (P1 ... P4) is capable of accessing the whole shared memory (Mem) directly, whereas on distributed memory systems each node only has direct access to its own memory and requires some mechanism to access remote memory.

Unlike for instance MPI, which usually uses two-sided communication, UPC is based on one-sided communication and targets *Direct Memory Access* (DMA) or *Remote Direct Memory Access* (RDMA) architectures respectively. UPC supports several high speed RDMA network architectures like Quadrics, InfiniBand or SCI.

Beside work-sharing and synchronization mechanisms, the main goal of UPC is to introduce the `shared` keyword to the C language, which allows programmers to define data as being shared between several parallel processes. As general abstraction of the two presented memory models, the elements of shared data structures like arrays are distributed over the nodes. The memory model of UPC is therefore called *distributed shared memory*. The nodes get some affinity to a 'local' part of the data, which is important to achieve high performance on distributed memory systems.

In UPC, shared data can transparently be accessed directly by each node; the UPC-compiler and -runtime take care of remote data requests dependent on the actual architecture. The benefit of UPC is that the same parallel UPC code can be compiled for different architectures without the necessity of architecture specific adaptations.

The memory consistency model of UPC defines two consistency types for shared data. Shared data can be accessed with *strict* or with *relaxed* consistency. The consistency type can be specified either globally by header files, locally with preprocessor macros, or individually for any shared variable.

The strict consistency model enforces completion of any memory access preceding a strict access, and delays any subsequent access until the strict access has been finished. Since this strict consistency model is expensive (i.e., causes high

latency) due to inter-node communication, it should only be used if necessary for correct program behavior.

Otherwise the relaxed model is appropriate: a sequence of memory accesses which are relaxed is allowed to be reordered as long as interdependencies are respected. Relaxed consistency allows the compiler and the runtime system to optimize memory access.

3 Cell Broadband Engine Memory Model

The Cell processor [3] is a hybrid multicore processor. The specification of the Cell Broadband Engine - the *Cell Broadband Engine Architecture* (CBEA) [4] - provides two different types of computation elements: a PowerPC-compliant *Power Processing Element* (PPE) and several Single Instruction Multiple Data processing cores, the *Synergistic Processing Elements* (SPE). These elements are connected via an on-chip high speed bus, the *Element Interconnect Bus* (EIB), and have shared access to an off-chip system memory.

Usually all computation-intensive tasks are dismissed to the SPEs while the administrative work (running the OS; coordination of the SPEs) is done by the PPE. In the remainder of this document we assume that a UPC program runs in parallel on several SPEs and that the PPE is not involved in any application-specific computation.

The processor elements and the EIB are shown in Fig. 3. The PPE itself consists of a computation unit - the *Power Processing Unit* (PPU) - and a transparent local cache. The SPEs have three major components:

- The *Memory Flow Controller* (MFC) enables access to the EIB - and thus the communication with the other processor elements.
- A relatively small *Local Store* (LS) holds the program code and the data of a particular SPE.
- The *Synergistic Processing Unit* (SPU) is the computation unit of the SPE.

The main memory is connected to the EIB via the *Memory Interface Controller* (MIC).

While the PPU has direct and transparent access to main memory, the SPUs have to instruct their MFC to perform a DMA-data-transfer from main memory to the LS. The data may not be accessed until the transfer has finished.

4 Mapping UPC to the Cell

The memory architecture of the Cell has two main features: first, there is a significant difference between the size of the large main memory (several GBytes) and the small LS (some 100kBytes); secondly, the main memory cannot be directly accessed by the SPUs.

In consequence, the memory architecture can not be seen as a classical shared memory architecture due to the missing direct memory access of all execution

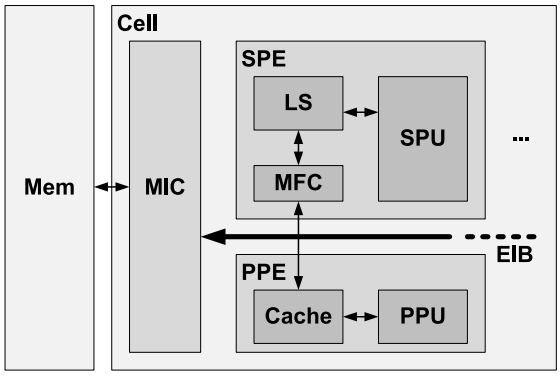

Fig. 3. Cell Broadband Engine Architecture

units. Furthermore, it is not strictly a distributed memory architecture, because the LS is negligibly small and not suited for distributed memory usage; thus all data have to reside in the shared main memory.

We propose the following abstraction to hide the specific memory architecture of the Cell and to provide easy access to shared data:

- The affinity of a UPC node to local data is no longer maintained, all data resides in the shared main memory.
- A runtime system has to be implemented, which handles the access to the 'remote' (in terms of not direct accessible) memory.

The runtime system has to load shared data into the LS on read access and to forward all write accesses to main memory while respecting the memory consistency model of UPC.

4.1 Runtime System

Aside from commercial UPC implementations by IBM and HP, there are several open UPC implementations, e.g., Berkeley UPC, GCC UPC (partly based on Berkeley UPC runtime), and MuPC from Michigan Tech. Because of the well documented interfaces and the existence of an UPC-to-C compiler of Berkeley UPC, we decided to base our UPC implementation for the Cell processor on Berkeley UPC.

Berkeley UPC consists of two components: the compiler, translating the UPC code into C code or an executable binary, and the runtime, supporting the execution of the UPC application and controlling shared memory access.

The UPC runtime itself is organized in layers. The top layer is the UPC application itself. The second layer is the UPC runtime which handles application initialization and shared memory management. This layer sits on top of the *Global-Address Space Networking* (GASNet) layer which serves as abstraction layer for the network interface.

The design of Berkeley UPC makes it easy for developers to extend the runtime system to new architectures by implementing a GASNet-provider for the particular architecture. Thus, the compiler and the runtime system do not need to be changed. This facilitates the adaptation for other platforms.

In the case of the Cell processor this procedure is not feasible due to the following reasons:

- The local storage of the SPEs is very small; the Berkeley UPC runtime is not optimized for memory consumption and occupies too much of the short memory. Furthermore, the Berkeley UPC runtime contains management code like application startup or memory allocation which is not required on the SPEs.
- The DMA access of the SPEs to main memory has some very strict rules concerning addresses and message size. Thus, the performance is highly reduced if these restrictions are not regarded by the runtime layer.

Therefore, our design omits the network abstraction layer (GASNet) and instead replaces the Berkeley UPC runtime system by a new system implementing the Berkeley UPC runtime API [5]. The management code is moved to the PPE, and only memory optimized code for data transfers is implemented for the SPEs.

4.2 Simple Approach

The simplest approach to implement UPC on the Cell is to forward any shared memory request directly to main memory. Any relaxed access can be implemented as simple DMA access. Relaxed reads interrupt program execution until the data have been transfered.

Strict accesses need some special attention, since all preceding relaxed transfers need to be finished and all following accesses need to be delayed until the strict transfer has finished. This can be done by marking the strict DMA access with a barrier tag and by using the synchronization mechanism of the Cell architecture (see [6, sec. 20]). This tag along with the synchronization ensures serialization of DMA transfers.

Data can not be transfered directly from main memory to any target address in local storage or vice versa, because source and target of any DMA transfer have to be aligned equally in a 16 byte grid. Since the Berkeley UPC-compiler uses a temporary variable to store remote values, the addresses will not match in most cases.

Furthermore, the DMA transfer size is restricted to 1, 2, 4, 8, or multiples of 16 bytes. This restriction is no problem for standard data types like `char`, `float` or `int`, but the size user-defined structures may not meet these requirements.

Finally, data must be aligned naturally, i.e., a transfer of 2 bytes must target an even address, a transfer of 4 bytes an address of a multiple of 4, and so on.

Due to these requirements, every DMA-transfer must address a buffer in local storage from which the actual data is copied to the target address.By using a buffer, the correct alignment of the data for DMA access (read or write) can be assured. In case that the transfer size matches the CBEA requirements, a

simple DMA transfer can be performed. In case of read access, the transfer size is enlarged to a valid value, and only the requested data is written from the buffer to the target address.

In case of write access, the transfer is portioned into several DMA transfers so that the size of each transfer has an allowed value. If, for example, 127 bytes have to be transferred, 5 transfers are performed: one transfer is necessary for $112 = 16*7$ bytes, and four additional transfers of 8, 4, 2, and 1 bytes respectively, summing up to 127 bytes.

This very simple, straight-forward approach, however, is inefficient due to the the following characteristics of the cell architecture:

- Every DMA transfer to main memory over the EIB will be split into naturally aligned atomic packages of 128 bytes. Even if only one byte is requested by the application, 128 bytes will be transfered and this single byte is selected by the DMA controller.
  Thus, sequential access to an array data structure will lead to long latencies caused by inefficient memory access - in worst case the same data is transfered again and again.
- A zero write solution from main memory to local storage and vice versa is impossible; this leads to further latency.

Therefore, we will present a software-managed cache as an improvement of these performance drawbacks of the simple UPC runtime implementation in the following section.

5 Optimization: Software Managed Cache

The UPC V1.2 specification [2] has some aspects which simplify the implementation of a per-node cache of remote shared data: as long as only relaxed shared data are read repeatedly, there is no need to reload previously touched data. Only when synchronization points are reached, a cache write back and flush is necessary. Synchronization points are all strict accesses to shared data, barriers, fences and locks [2, 7].

Consequently, subsequent relaxed accesses to shared data can be cached in the LS of the SPEs until a synchronization point is reached and the cache needs to be flushed. The software-managed cache holds cache lines with a size of 128 bytes - reflecting the DMA transfer size of 128 bytes. The effects of an enlargement of the cache line size will be considered in future research.

Shared data can be accessed either by a read- or write-operation. In the following paragraphs, we will propose caching strategies for these operations.

5.1 Read-Operations

The first step to handle a read access to relaxed shared data is to query the local cache for the cache line containing the data. Only if the data is not found, a blocking DMA transfer from main memory is necessary. In the last step, the data is returned from the cache.

5.2 Write-Operations

While the algorithm to handle read accesses is rather simple, write accesses can be implemented in several ways with varying impacts on the performance. Therefore we propose three algorithms and discuss their advantages and disadvantages in the following paragraphs:

Direct Write Through. Like in the read-case, a cache lookup is performed at first. If the touched cache line resides in the cache, the data is copied to that cache line. If the cache line is not found in the cache, a cache frame is picked from the cache as buffer and the data is copied to it. The cache frame is marked as write-only since it does not contain the whole data of the cache line. In both cases, a DMA transfer to main memory is started after the data was copied to the cache. Note that this transfer is non-blocking. The cache frame is marked as busy. At a future call to the runtime system, the end of the DMA transfer can be detected and the busy-tag be reset.

This method has two main advantages: the administrative overhead is rather small and the non-blocking DMA write allows further computation in parallel to the data transfer.

The disadvantage is that the bus and the memory interface may get flooded with writes. As described before, for each transfer 128 bytes are moved, which results in an unacceptable overhead if memory is traversed sequentially and the same cache line is hit several times.

Repeated transfers of the same cache line can be prevented by the following approach:

Bundled Writes With Cache Line Read. On a write access to relaxed shared data a cache lookup is performed. If the cache line resides in the cache within a certain frame, data is transfered to this cache frame. If the cache line is not found in the cache, a DMA request from main memory for the cache line is issued and the data is copied after the transfer has finished. Note that this DMA transfer is blocking.

After the data has been copied to the cache, the cache line is marked dirty, and a *delayed write entry* is added to a ring buffer structure. Writes to the same cache line are merged to reduce memory traffic. Writes may be flushed when the ring buffer gets filled up; the whole cache is flushed when a synchronization point is reached.

If a dirty cache line needs to be replaced due to cache collisions, the corresponding entry in the ring buffer needs to be invalidated.

This variant removes some pressure from the EIB and the memory controller since it is able to merge sequential or neighboring writes, but it adds some significant latency to the first cache hit of a write. This latency is compensated if data from the cache line are read later on, otherwise this delay is completely unnecessary.

Unnecessary DMA reads are avoided by our third approach:

Bundled Writes With Cache Line Read On Demand. This version delays the cache line read latency until an actual read is initiated. In contrast to the preceding approach, the cache line is not read from main memory if a relaxed write hits a cache line that does not reside in cache. The data is just copied to an unoccupied cache frame and an entry is added to the ring buffer. The cache line is marked as write only.

If a relaxed read hits a cache line which is marked as write only, the cache line is read from memory to a free cache frame and the written data is copied to this line. The reference in the ring buffer entry is updated and the old cache line discarded.

This approach adds some further effort to relaxed reads in case the cache line is found with a write only tag in the cache but an acceptable performance can be expected in most use cases.

The Direct Write Through approach has the main benefit that the administrative overhead and thus the code size is rather small which is important due to the very small LS. The other two approaches need more code and administrative data but may result in a significantly higher memory performance since the number of issued DMA transfers is reduced.

Furthermore, by using Direct Write Through, the write accesses are mostly distributed over time; several SPEs will access memory in a quasi randomized fashion which reduces the competition of the SPEs for the system bus and the memory controller. The other approaches flush the cache at synchronization points. In case of strict accesses, fences and locks (local synchronization), we expect this to turn out to be random, too. If the application is fine-grained-synchronized by barriers (global synchronization), a hard competition at most cache flushes might occur - leading to high latencies.

To evaluate the impact of the cache algorithms, all three approaches will be implemented and compared in future research.

5.3 Cache Flush

In case of a strict memory access (read or write), the cache is flushed, i.e., all cache lines which are marked dirty are written to main memory and all cache frames are marked empty.

In contrast to classical cache architectures where concurrent access to the same cache line by multiple processing units is handled by a strict cache coherence protocol, relaxed accesses to shared data leave it to the programmer to assure that no conflicting writes to the same data are performed by the SPEs. This removes the necessity of the implementation of a coherence protocol, but requires special precautions in case of cache flushes.

Classical cache architectures simply write whole cache lines back to main memory. Our software managed cache is not allowed to transfer any bytes to main memory which were not actually changed by a write. By this we ensure that no valid changes by another SPE to data in the same cache line are overwritten.

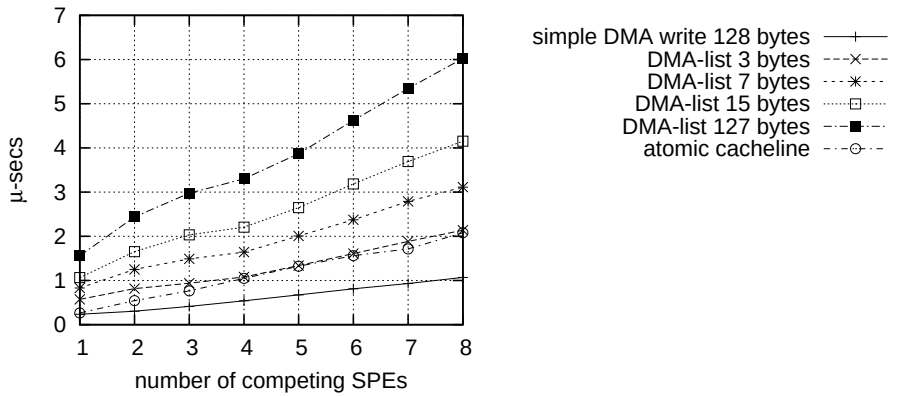

Fig. 4. Measurement of Competing Writes to a Cache Line

Therefore a bit-field needs to be stored for each dirty cache line, indicating exactly which bytes have been changed.

One way to transfer only the dirty bytes to main memory is the method described in section 4.2. Another solution is to perform an atomic access to main memory: the Cell processor allows a cache line to be read from main memory to a local buffer with support of an atomic unit [6, sec. 6.2.2]. The dirty bytes are written from local cache to the buffer, and the modified buffer is written back to main memory only if the cache line in main memory was not in the meantime modified by another SPE. Otherwise the cache line has to be reloaded from main memory and another attempt has to be made.

Fig. 4 shows measurements of up to 8 competing SPEs writing to the same cache line in main memory with multiple DMA transfers and the atomic cache mechanism. The results show, that the fastest way to update a cache line is to make a single transfer of 128 bytes to main memory. This however is only possible, if the whole cache line was modified by a relaxed write. If only some data has been modified, the fastest update can be performed with the atomic feature of the CBEA which allows for a really fast read and write to a cache line.

Therefore, our implementation will use the atomic unit for flushing not completely dirty cache lines.

5.4 Impacts on Performance

Remote data caching for UPC [7, 8] and a software managed cache for Cell applications [9] have been shown to positively influence sequential data access; random access is slightly slowed down by the additional overhead. Thus, we also expect similar effects from our data cache implementation on the relaxed access of shared data.

6 Conclusion

The goal of this paper was to evaluate the portability of UPC to the Cell processor. We showed, that an elegant mapping of the UPC memory model to the Cell is feasible. Some special characteristics of the Cell architecture have to be considered, but the main abstractions of UPC can be transferred without loss of functionality. The memory model of UPC is transfered to the Cell architecture by keeping shared data in main memory and transferring chunks of data to the local storage of the computing units and vice versa on demand. We introduced a software managed cache to reduce memory latencies for memory accesses. Several possibilities of cache implementations were described; these will be implemented and evaluated in future research.

The limited local storage of the computation units has to store data as well as program code. Our proposal so far does not allow existing UPC applications, which exceed this limit, to be ported to the Cell processor. Thus, compiler-enabled partitioning of program code and caching not only of data but also of code will be necessary [10, 11]. We plan to integrate a solution for code partitioning in our UPC implementation for the Cell processor. Furthermore, usage of UPC should not be restricted to stand-alone Cell systems. By bringing UPC to Cell clusters, UPC applications will profit from higher parallelization. Thus, we will expand our research to Cell clusters in the future.

Our next task will be the implementation of the presented algorithms.

References

1. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.: UPC: Distributed Shared-Memory Programming. Wiley-Interscience (2003)
2. UPC-Consortium: UPC Language Specifications V1.2. [`http://upc.gwu.edu`] (2005)
3. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. **49** (2005) 589–604
4. IBM: Cell Broadband Engine Architecture, Version 1.02. (October 2007)
5. Bonachea, D.: The Berkeley UPC Runtime Specification V3.9. [`http://upc.lbl.gov/docs/system/upcr.txt`] (2002)
6. IBM: Cell Broadband Engine Programming Handbook, Version 1.1. (April 2007)
7. Zhang, Z.: A performance model for unified parallel c. PhD thesis, Michigan Technological University (2006)
8. Zhang, Z., Savant, J., Seidel, S.: A upc runtime system based on mpi and posix threads, IEEE Computer Society (2006) 195–202
9. Balart, J., Gonzalez, M., Martorell, X., Ayguade, E., Sura, Z., Chen, T., Zhang, T., O'Brien, K., O'Brien, K.: A novel asynchronous software cache implementation for the cell-be processor. In: The 20th International Workshop on Languages. (2007)
10. Eichenberger, A.E., et. al.: Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. IBM Syst. J. **45** (2006) 59–84
11. Eichenberger, A.E., et. al.: Optimizing compiler for the cell processor, IEEE Computer Society (2005) 161–172