Runtime support for OpenMP language on Cell Broadband Engine

Guansong Zhang, Raul Silvera, Mark Mendell*

IBM Toronto Lab
Toronto
ON, L6G 1C7, Canada

Abstract:

The Cell processor has many advantages: computation power, memory bandwidth, and power consumption. On the other ahdn, it is hard to program. The two distinct ISAs, limited SPU address space and explicit DMA communications, all complicate the programming effort on this processor. We are helping this situation by providing an OpenMP compiler for Cell. We can implement a full OpenMP compliant compiler to run potentially any OpenMP programs on such a processor with reasonable performance. In this paper, we introduce the runtime support for the OpenMP compiler and show how to map major OpenMP constructs onto a Cell processor. This paper will help programmers write Cell applications using industry standard OpenMP, and help people with an OpenMP background use the Cell processor more effectively.

Keywords: Cell, OpenMP, parallel region, workshare, barrier

1 Introduction

The Cell Broadband Engine ^{TM} (Cell BE) processor [1] is now commercially available in both the Sony PS3 game console and the IBM BladeCenter which represents the first product on its development road-map. The anticipated high volumes for this non-traditional commodity hardware continue to make it interesting in a variety of different application spaces, ranging from the obvious multi-media and gaming domain, through the HPC space (both traditional and commercial), and to the potential use of Cell as a building block for very high end supercomputing systems [2].

This first generation Cell processor provides flexibility and performance through the inclusion of a 64-bit multi-threaded Power Processor ^{TM} Element (PPE) with two levels of globally-coherent cache and support for multiple operating systems including Linux. For additional performance, a Cell processor includes eight Synergistic Processor Elements (SPEs), each consisting of a Synergistic Processing Unit (SPU), a 256K local memory, and a globally-coherent DMA engine. Computations on the SPUs are performed by 128-bit wide Single Instruction Multiple Data (SIMD) functional units. An integrated high bandwidth bus, the Element Interconnect Bus (EIB), glues together the nine processors and their ports to external memory and IO, and allows the SPUs to be used for streaming applications [3].

* The opinions expressed in this paper are those of the authors and not necessarily of IBM.

Data is transferred between the local memory and the DMA engine [4] in chunks of 128 bytes. The DMA engine can support up to 16 concurrent requests of up to 16K bytes originating either locally or remotely. The DMA engine is part of the globally coherent memory address space; addresses of local DMA requests are translated by a Memory Management Unit (MMU) before being sent on the bus. Bandwidth between the DMA and the EIB bus is 8 bytes per cycle in each direction. Programs interface with the DMA unit via a channel interface and may initiate blocking as well as non-blocking requests.

Programming the SPE processor is significantly enhanced by the availability of an optimizing compiler which supports SIMD intrinsic functions and automatic simdization [5]. However, programming the entire Cell processor consisting of the coupled PPE and 8 SPE processors is a much more complex task, requiring partitioning of an application to accommodate the limited local memory constraints of the SPE, parallelization across the multiple SPEs, orchestration of data transfers through insertion of DMA commands, and compiling for two distinct ISAs.

To help users to harness such a powerful yet complicated processor and to port existing legacy code, we developed an OpenMP[6] implementation for the system. This is the same as a typical OpenMP implementation such as in Figure 1.

In this figure, the language-dependent frontends will translate user source code to an *intermediate representation (IR)*, to be processed by an optimizing compiler into a resulting *node program*, which interacts with a *runtime* environment that starts and controls the multi-threaded execution.

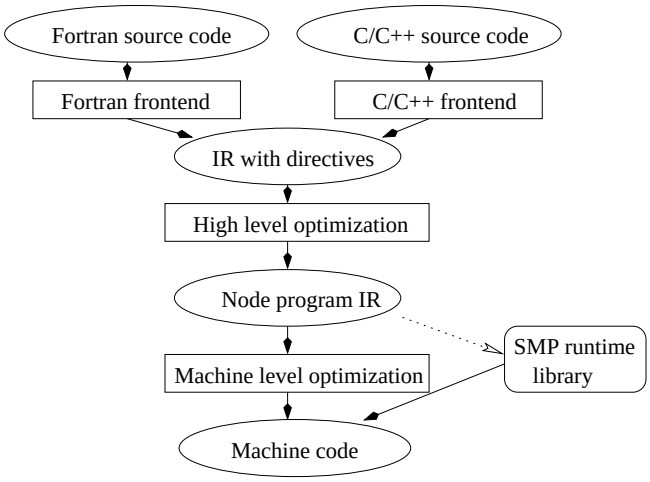

Fig. 1. Compiler and runtime structure

With this system, the programming the Cell processor can benefit from an industry standard in a familiar user environment. The compiler will:

- **Systematically manage multiple SPU threads** Users do not need to know Cell SDK details, as thread management on the SPUs is provided through the OpenMP framework.
- **Automatically overlap computation and communication** The compiler will generate data communication between the PPU main memory and the SPU local store. DMA buffering and software cache can be automatically used to exploit the large bandwidth between main memory and all 9 processors.
- **Enlarge the limited SPU address space** Our OpenMP Cell compiler will treat SPU local storage more like a cache in a traditional processor. Large application data and code will be kept in the main memory of the PPU and paged in or transfered as appropriate.

There are already papers introducing the second and the third points[7] of the system, but little discussion has been dedicated to the OpenMP runtime environment itself. The OpenMP standard continues to evolve and there are still many open issues in this area, including the mechanisms required to support nested parallelism, task, and more.

In this paper, we will focus on the first item, the major runtime environment implementation itself[1].

The runtime system we have developed is based on the OpenMP runtime library we use for other XL series compilers, including C/C++ and Fortran on AIX©and Linux systems [8]. This paper shows how to map the OpenMP environment to the Cell system. It concentrates on discussions of the implementation considerations, rather than implementation details. Some performance numbers are also given.

2 Parallelism in OpenMP

The OpenMP standard is jointly defined by a group of major computer hardware and software vendors[6]. This makes it a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer. The most powerful semantics in the standard is represented by a *parallel construct*. We will first look at mapping a parallel construct onto a Cell processor.

2.1 A `parallel` region

There are two types of processors in a Cell chip. The first type of processor element, the PPE, is a 64-bit PowerPC Architecture core. It is fully compliant with the 64-bit PowerPC Architecture and can run 32-bit and 64-bit operating systems and applications. The second type of processor element, the SPE, is optimized for running compute-intensive applications. The SPEs are independent processors, each running its own individual application program. Each SPE has full access to coherent shared memory through DMA, including the memory-mapped I/O space. The designation *synergistic* for this processor was chosen carefully because there is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system, and, in many cases, the

[1] Topics such as threadprivate and C++ object privatization will be discussed elsewhere.

top-level control thread of an application. The PPE depends on the SPEs to provide the bulk of the application performance. This dependency is reflected in our OpenMP standard implementation: the sequential part of the program will be executed by the PPU, and the parallel part, i.e., `parallel` regions, of the program will be executed by the SPUs. In this way, the user program can take full advantage of the variety of PPU system libraries, while the SPEs will help when the program is within the computationally intensive part.

This implementation scheme brings some challenges to the compiler system, including: compiling for both PPU and SPU, switching between PPU and SPU execution, separating PPU libraries from SPU ones, and more. We will cover some of these topics in this paper.

From a memory system point of view, the OpenMP standard specifies a relaxed-consistency, shared-memory model. This model allows each thread to have its own temporary view of memory. A value written to a variable, or a value read from a variable, can remain in the thread's temporary view until it is forced to share memory by an OpenMP flush operation. This memory model can be efficiently implemented on the Cell memory structure, even though each SPE has just 256K of directly accessible local memory for code, data, stack and heap. In order to handle such limited memory, we only allocate private variables accessed in SPE code in the SPE local store. Shared variables reside in system memory, and SPE code can access them through DMA operations using either a DMA buffer mechanism or software cache.

We set up the outermost parallel region by specifying the starting thread and slave threads on the PPU and the SPUs separately. Nested parallel regions will be executed on SPUs only. We still honor OpenMP data scoping clauses, using the encountering threads to execute the nested parallel region sequentially.

2.2 The `if` clause

The `if` clause on a `parallel` region is useful when it is not known at compile time that the computation part in a `parallel` region is intense enough to outweigh the overhead of communication involved. For example, the following code will be parallel executed on SPUs only when we have a large enough loop bound.

```
#pragama omp parallel for if (n>LIMIT)
for (int i=0; i<n; i++) {
  compute(i);
}
```

This could be rewritten as, (although it is equivalent in OpenMP 2.5, it may not be true considering nested parallel regions in OpenMP 3.0)

```
if (n>LIMIT) {
  #pragama omp parallel for
  for (int i=0; i<n; i++) {
    compute(i);
  }
} else {
```

```
      compute(i);
    }
```

to avoid overhead for starting up a parallel region. The function `compute` may be executed on either the PPU or multiple SPUs. In such a case, the compiler have to *clone* the procedure for two different ISAs. We need to redraw the Figure 1 as Figure 2.

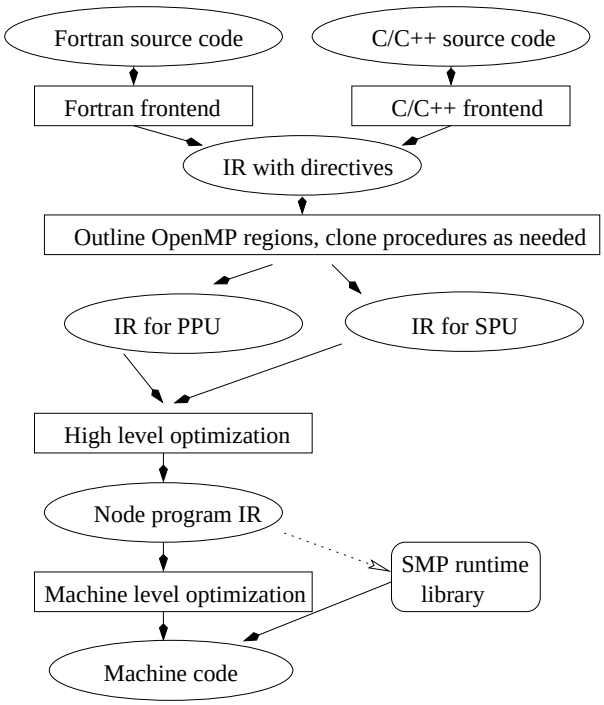

Fig. 2. Clone for two ISAs

After we outline the parallel sections into procedures for an OpenMP region, call graph analysis is needed within the compiler to trace which functions can be reached from PPU and/or SPU. We then need to *clone*, make two versions of, the routine if a procedure will be reached by both PPU and SPU sides. The SPU code is generated with the SPU linker, as a single SPU program executable. The PPU code is also generated with the SPU program *embedded* and linked into the final PPU executable.

2.3 The `num_threads` clause

According to the OpenMP standard, the thread which first encounter a parallel construct becomes the master thread of a new team, with the thread number as zero for the dura-

tion of the `parallel` region. Initially we did put the PPU part of the parallel region. Experiments later on have shown some drawbacks with this approach.

- The difference in computation power between a PPU and an SPU makes the PPU thread less useful in a workshare of the `parallel` region. We can not see any benchmarks benefit from assigning partial work to a PPU thread. The PPU thread may actually slow down the application, with load balancing playing a role.
- In order to solve the above problem, we have to apply special loop schedules to disable the PPU inside an OpenMP workshare. This conflicts with the basic idea in an OpenMP environment, i.e., the threads in a parallel region, including both the master and the slave threads, are as equal as possible.
- Even when we don't have any workshare in a parallel region, the PPU participation means that the computation in the parallel region will be carried out by both PPU and SPU, and may have different floating point results. SPU single precision arithmetic is not fully IEEE-794 compliant.
- If the PPU participates in a parallel region, it will force the compiler to clone all the procedures can be reached by a parallel region, even without an `if` clause. This will extend the compilation time and may cause certain optimizations, such as pointer analysis, to give up due to an artificially enlarged problem size.
- This will also prevent a user from providing his/her own computation kernel in an SPU only format. If parallel regions are SPU only, an advanced user can compile a procedure with a software-cache-aware SPU compiler and link it into the OpenMP compiler generated code, as long as this procedure can only be reached from a parallel region (without an `if` clause).

We decided to have the PPU thread start a `parallel` region, then suspend execution and resume after the `parallel` region. Inside the `parallel` region, the `num_threads` clause or OpenMP internal control variables will be used to determine how many SPU threads will participate, and one of them will be picked as the master thread. This is more in keeping with the upcoming OpenMP 3.0 [9] task view, where the code outside and inside a parallel region are considered as different tasks.

We also set up a upper limit for the number of SPU threads can be used, it will be the maximum number of SPUs available on the system. The concept is the same as the `OMP_THREAD_LIMIT` environment variable in OpenMP 3.0.

3 Sharing work through workshares

When a `parallel` region starts, multiple threads will execute the code inside the region concurrently,

In Figure 3, a master thread starts up the `parallel` region executed by 8 threads. Through parallel regions, multiple threads accomplish *worksharing* in an OpenMP program.

The simplest format of worksharing is replicated execution of the same code segment on different threads. It is more useful to divide work among multiple threads — either by having different threads operate on different portions of a shared data structure, or by having different threads perform entirely different tasks. Each cooperation

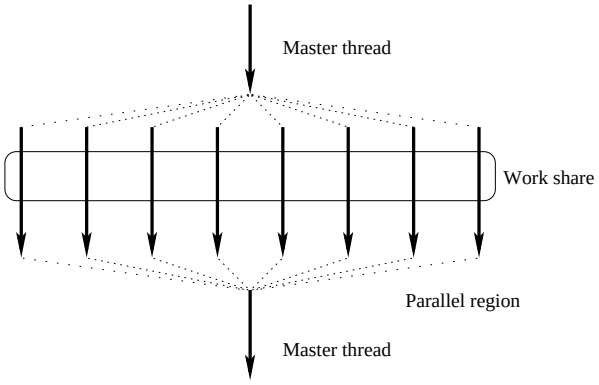

Fig. 3. Parallel region

of this kind is considered as a *workshare*[2] in this paper. In the Figure 3, we use a round-cornered rectangular block to represent one workshare.

The most common forms of workshares in OpenMP specification are `omp for` and `omp sections`,

We can also consider OpenMP `single` as a special workshare too. A `single` construct is semantically equivalent to a `sections` construct with only one `section` inside. For a `single` construct, the first thread that encounters the code will execute the block. This is different from a `master` construct, where the decision must be made by checking the thread id.

All the workshares introduced here have an important common feature: they have an implicit barrier at the end of the construct, which may be disabled by using the optional `nowait` clause. Besides, An explicit `barrier` region is semantically equivalent to an empty workshare without the `nowait` clause.

This observation reveals that the barrier is the most important workshare to in a runtime supporting system. We will discuss more about a barrier implementation in the next section.

Different implementations may have a different kind of classification. The real advantage of considering them all as workshares is for simplifying the developing process. From an implementation point of view, the common behaviors will lead to a common code base, thus improving the overall code quality. And once we have a barrier implementation on Cell, the implementation of the rest workshares are relatively easier.

[2] The concept of workshare here is different from the parallel construct, `WORKSHARE` in Fortran OpenMP specification 2.0, which can be treated semantically as the combination of `for` and `single` constructs.

4 The `omp barrier`

Figure 4 shows a simple barrier that can be used for a OpenMP runtime system. Each arriving thread will decrease the counter with an atomic operation, and they will spin checking the counter value and not leave the barrier until the counter becomes zero.

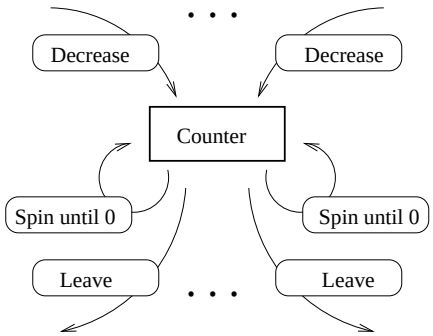

Fig. 4. Barrier implemented with fetch-and-add

Algorithm 1 describes a more realistic implementation used in the XL OpenMP runtime system on AIX and Linux,

Instead of having one counter, each thread will operate its own counter, thus the expensive fetch and add operation can be replaced with a regular add. In order to further reduce the contention, we also have a local sensor to reduce the false sharing between reads from different threads[10].

The algorithm can be mapped perfectly onto the Cell processor. The distributed counter and local sensor can be replaced by the SPU output and input mailbox separately. For example, the following code executed on an SPU and achieve the algorithm drawn in Figure 5.

```
{
  int data;
  spu_writech(SPU_WrOutMbox, 1);
  data = spu_readch(SPU_RdInMbox);
}
```

And the PPU version,

```
unsigned int data;
spe_context_ptr_t id = _xlbesplitSPUThreads[i].spuid;

for (int i=0; i<num_threads; i++) {
  while(spe_out_mbox_status(id) == 0);
  spe_out_mbox_read(id, &data, 1);
}
```

Algorithm 1: Barrier with distributed counter and local sensor

```
Data : Distributed counter with each element as one
Data : Local sensor with each element as one
begin
    Decrease my own distributed counter element;
    if I am the master thread then
        repeat
            foreach element in distributed counter do check if it is zero
        until all distributed counter elements are zero;
        foreach element in distributed counter do
            set it back to one
        end
        foreach element in local sensor do
            set it to zero
        end
    else
        repeat
            check my local sensor element
        until it is zero;
    end
    Set my own local sensor element back to one;
end
```

```
  for (int i=0; i<num_threads; i++) {
    while (spe_in_mbox_status(id) == 0);
    spe_in_mbox_write(id, &data, 1, SPE_MBOX_ANY_NONBLOCKING);
  }
```

In the real implementation, a barrier is also a chance for PPU and SPU to exchange data such as for handling single, reduction, copyprivate etc.

5 Benchmarks and Summary

In this section, we use some benchmark numbers to summarize the project. The OpenMP compiler introduced in [7] and this paper as a product are general available now. The data listed here can be reproduced, although some slight difference may exit, as we are using a development version of the compiler.

As the Fortran support is not robust as the C/C++ yet, We choose the NAS 2.3 C version as our benchmarks.

Table 1 shows the performance of 8 SPUs against one PPU execution. We also include two extra benchmarks, a PDE solver, packaged in the single source compiler distribution as an example, and LBM, the CPU benchmark in spec2k6 with OpenMP directives.

We do need to modify the source code a little bit to make it fit for the compiler, the main changes are

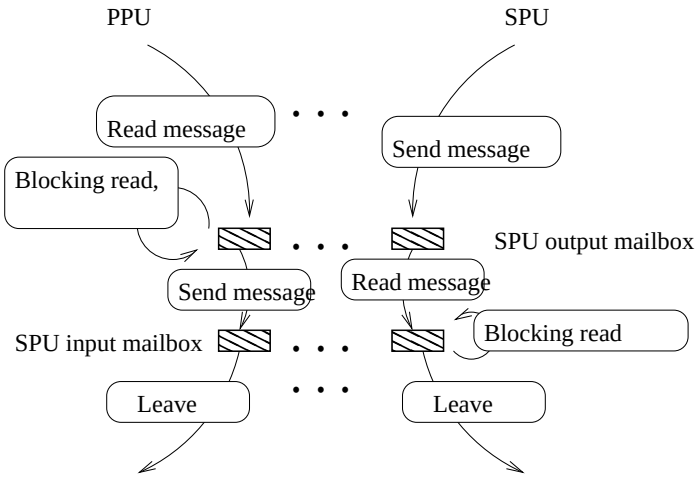

Fig. 5. Barrier with SPU output and input mailbox

- Using restricted pointer or adding disjoined pragma to indicate that two arrays will not overlay each other. Although interprocedure compilation can trace pointer targets, there are cases which do need users to give more accurate info.
- Change the data location, either privatize a variable to make it stay on SPU local store or malloc an array from system storage. These two looks contradictory, the first one aims to speed up access, the second one is to save SPU local store space.
- Data access alignment. This is due to DMA operation will be more efficient if the data can be 128 byte aligned.

From the table we can see that not every benchmark speed up nicely. Here are many reasons for this. A complicated parallel region with a lot of branches inside may cause SPU slow down. Data access pattern is not regular, DMA tiler can not tile the loop, so the memory access has to go though software cache. As we are improving the tiler performance, we do hope some of them can be further improved.

In this paper, we summarized how to map OpenMP constructs onto a Cell processor, from a `parallel` region to different workshares and barriers. We have omitted the implementation details of these, concentrating on the framework which will help the user visualize the framework of the OpenMP runtime system on a Cell processor. Please refer to the references for how to implement software cache DMA tiler and SPU simdization.

There is still on-going work to further enhance the compilation system, such a enlarge the DMA tiler region, improving auto simdization. For future runtime enhancements, we will look at the task support in the OpenMP 3.0 standard.

Benchmark	# of lines changed	sequential time	parallel time	speedup
BT	0	588.48		
CG	0	20.12	13.78	1.46
EP	0	72.05	12.63	5.66
FT	4	42.19	11.66	3.65
IS	10	10.54	2.68	3.94
LU	0	356.8	209.71	1.71
MG	10	17.38	2.41	7.2
SP	10	413.99	134.69	3.08
PDE		27.59	0.65	42.38
LBM		1553.91	147.61	10.66

Table 1. Performance evaluation

6 Acknowledgments

Thanks to Tao Zhang from IBM Watson research lab and Daniel A. Brokenshire from IBM Austin Quasar/Cell software development for the discussions on the barrier implementation on the Cell system.

7 Trademarks and copyright

References

1. D. P. et al. The design and implementation of a first-generation cell processr. In *IEEE International Solid-State Circuites Conference (ISSCC)*, February 2005.
2. Samuel Williams. et al. The potential of the cell processor for scientific computing. In *Conference on Computing Frontiers*, 2006.
3. M. Gordon. et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.
4. M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication net-work: Built for speed. *IEEE Micro*, 26(3), 2006.
5. A. E. et al. Vectorization for simd architecture with alignment constraints. In *Conference on Programming Language Design and Implementation (PLDI)*, 2003.
6. OpenMP Architecture Review Board. Openmp application program interface version 2.5, 2005. http://www.openmp.org.

7. Kevin O.Brien, Kathryn O.Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. In *International Workshop on OpenMP (IWOMP)*, 2007.
8. Guansong Zhang, Raul Silvera, and Roch Archambault. Structure and algorithm for implementing OpenMP workshare. In *WOMPAT*, Lecture Notes in Computer Science. Springer, 2004.
9. OpenMP Architecture Review Board. Openmp application program interface version 3.0, public review, 2008. http://www.openmp.org.
10. Guansong Zhang et al. Busy-wait barrier synchronization with distributed counter and local sensor. In Michael Voss, editor, *WOMPAT*, volume 2716 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2003.