AN MPI PERFORMANCE MONITORING INTERFACE FOR CELL BASED COMPUTE NODES

HIKMET DURSUN,[1,2] KEVIN J. BARKER,[1] DARREN J. KERBYSON,[1] SCOTT PAKIN[1]

[1]*Performance and Architecture Laboratory (PAL), Los Alamos National Laboratory Los Alamos, New Mexico 87545, USA*

RICHARD SEYMOUR,[2] RAJIV K. KALIA,[2] AIICHIRO NAKANO,[2] PRIYA VASHISHTA[2]

[2]*Collaboratory for Advanced Computing and Simulations, Department of Computer Science University of Southern California, Los Angeles, California 90089-0242, USA*

ABSTRACT

In this paper, we present a methodology for profiling parallel applications executing on the family of architectures commonly referred as the "Cell" processor. Specifically, we examine Cell-centric MPI programs on hybrid clusters containing multiple Opteron and IBM PowerXCell 8i processors per node such as those used in the petascale Roadrunner system. We analyze the performance of our approach on a PlayStation3 console based on Cell Broadband Engine—the CBE—as well as an IBM BladeCenter QS22 based on PowerXCell 8i. Our implementation incurs less than 0.5% overhead and 0.3 μs per profiler call for a typical molecular dynamics code on the Cell BE while efficiently utilizing the limited local store of the Cell's SPE cores. Our worst-case overhead analysis on the PowerXCell 8i costs 3.2 μs per profiler call while using only two 5 KiB buffers. We demonstrate the use of our profiler on a cluster of hybrid nodes running a suite of scientific applications. Our analyses of inter-SPE communication (across the entire cluster) and function call patterns provide valuable information that can be used to optimize application performance.

Keywords: Application performance, profiling, cell processor

1. Introduction

Application developers at the forefront of high-performance computing (HPC) have been investigating the use of hybrid architectures to improve application performance. Hybrid architectures attempt to improve application performance by combining conventional, general-purpose CPUs with any of a variety of more specialized processors such as GPUs, FPGAs, and Cells. The complexity stemming from hybrid architectures make understanding and reasoning about application performance difficult without appropriate tool support.

In this paper, we present a profiling library that can trace not only intra-Cell direct memory access (DMA) events but also inter-Cell message passing. Our implementation is efficient in terms of resource consumption (only 12 KiB of SPE local store memory is required) and has an overhead of less than 0.3 μs per profiler call for a typical scientific application executing on the Cell BE.

We employ a *reverse acceleration* programming model in which the hybrid cluster architecture is presented to the programmer as a logical cluster of Cell SPE processors by using the Cell Messaging Layer (CML) [1]. CML significantly reduces the effort needed to port applications to Cell clusters and has been used to port several scientific applications (e.g., the Sweep3D deterministic particle-transport kernel) to Los Alamos National Laboratory's petascale Roadrunner supercomputer (comprising 6,120 dual-core Opterons plus 12,240 PowerXCell 8i processors). CML provides a subset of the functions and semantics of the MPI standard [2] including point-to-point communication, broadcasts, barriers, and global reductions.

The Cell processor's complex architecture—eight *synergistic processing elements* (SPEs) managed by a single *power processor element* (PPE)—makes profiling tools essential for performance optimization. Traditional tools merely monitor performance events on PPEs, which provide less than 6% of PowerXCell 8i flops performance and are usually used for solely controlling SPE processes instead of computing. The IBM Cell Software Development Kit (SDK) [3] includes a Cell performance-debugging tool (PDT) that helps analyze the performance of a single Cell board (up to two Cell processors) with two PPEs that share the main memory, run under the same Linux operating system, and share up to 16 SPEs. PDT can trace only a specific set of SDK library functions such as SPE activation, DMA transfers, synchronization, signaling, and user-defined events. Because PDT involves the slow PPE on the critical path of tracing, the PPE can easily become a performance bottleneck and may even influence application performance. Another tool for analyzing Cell performance is Vampir [4], which Nagel et al. used to visualize intra-Cell events such as mailbox communication and DMA transfers [5].

The key difference between our work and the works mentioned above is that we perform cluster-level analysis for MPI programs running on compute nodes featuring a hybrid architecture comprising AMD Opterons/PowerXCell 8i processors and PlayStation3 (PS3) commercial gaming consoles featuring Cell BE processors. The underlying message-passing model of CML, which treats an entire cluster of Opterons+Cells (or PS3s) as a homogenous collection of SPEs, has a central importance to our cluster-wide analysis. In addition to monitoring the same types of intra-Cell events as existing Cell profilers, our implementation can log inter-Cell, inter-blade, and inter-node communication. We have tested our implementation on up to 256 SPEs, although there is nothing limiting us from scaling up to thousands or even tens of thousands of SPEs.

Two parallel scientific applications—lattice Boltzmann (LB) flow simulation and atomistic molecular dynamics (MD) simulation—are used to test the profiler on the PS3 and hybrid Opteron+Cell Roadrunner architecture using CML. Two sample uses of the profiler are also demonstrated: communication analysis and call-stack analysis.

The organization of the rest of this paper is as follows: Section 2 provides information about the Cell architecture, the Cell Messaging Layer, and our experimental testbed. Section 3 discusses the software design and implementation of our profiler software. Section 4 analyzes profiler performance using microbenchmarks and some sample applications. Finally, we summarize our study in Section 5.

2. Architectural Background and Testbed

In this section we describe the architecture of the Cell Broadband Engine and PowerXCell 8i that provides the bulk of the performance of our target cluster and the focus of our profiler study. We then briefly summarize the overall architecture of our testbed. Finally, we describe the Cell Messaging Layer, which is an enabling technology for exploiting hybrid (or completely cell based) clusters and therefore a key insertion point for profiler events.

2.1. *Cell Broadband Engine and IBM PowerXCell 8i*

Cell BE has a heterogeneous architecture incorporating a power processor element (PPE) and eight synergetic processing elements (SPEs) on the same chip. SPEs are connected via an element interconnect bus (EIB), which supports a peak bandwidth of 204.8 GB/s for intra-chip data transfers among the PPE, SPEs, the memory, and the I/O interface controllers [6]. A single Cell BE has a peak single-precision performance of 217.6 Gflops/s for which it took attention of the high performance computing community in the recent years [7], whereas its double-precision peak is limited to 21 Gflops/s.

The IBM PowerXCell 8i (also referred as the Cell extended Double-Precision, Cell-eDP) is the latest implementation of the Cell BE featuring 108.8 Gflops/s on double-precision operations. It drives the fastest supercomputer at the time of this writing, Roadrunner at Los Alamos [8]. Each SPE of PowerXCell 8i contains a 3.2 GHz synergetic processing unit (SPU) core, 256 KB of a private, program-managed local store (LS) in place of a cache, and a memory flow controller (MFC) that provides DMA access to main memory. The SPE uses its LS for efficient instruction and data access, but it also has full access (via DMA) to the coherent shared memory, including the memory-mapped I/O space.

To make efficient use of the EIB and to interleave computation and data transfer, the PPE and 8 SPEs are equipped with a DMA engine. Since an SPE's load/store instructions can access only its private LS, each SPE depends exclusively on DMA operations to transfer data to and from the main memory and other SPEs' local memories. The use of DMAs as a central means of intra-chip data transfer maximizes asynchrony and concurrency in data processing inside a Cell processor [9].

2.2. *Testbed*

The PS3 features an identical Cell BE to the ones in IBM BladeCenter QS20. Recently the gaming console has been used as a low-cost computing platform by scientists [10]. However, one of the SPEs is disabled in PS3s for chip yield reasons and another SPE is reserved for use by GameOS operating system which acts as a hypervisor, and virtualizes the system resources. Out of 256 MB Rambus Extreme Data Rate (XDR) memory on PS3, only 200 MB is accessible to Linux OS and applications. Even though PS3s are not crafted for high performance cluster computing [11], they offer a valuable testing platform for tools targeting Cell based architectures. In this paper we use a PS3 console to quantify the overhead that our profiling library incurs.

Our second testing platform comprises 8 nodes, called tri-blades, where each tri-blade has two IBM QS22 Cell blades and one IBM LS21 AMD Opteron blade. The QS22 contains two PowerXCell 8i processors running at 3.2 GHz and each with an associated 4 GB of DDR2 memory. The LS21 blade includes two dual-core Opteron cores clocked at 1.8 GHz. Each tri-blade has a single connection to a Mellanox 4x DDR InfiniBand network. Typically, the Opterons handle mundane processing (e.g., file system I/O) while mathematically intensive elements are directed to the Cell processors. Each tri-blade in our testbed is architecturally identical to the tri-blades used in Roadrunner.

2.3. *Cell Messaging Layer*

CML is an implementation of common MPI functions for SPE-to-SPE communication in Cell-based clusters. The programming model underlying the CML is that applications run entirely on the SPEs. The SPE-centric model of CML assigns unique MPI ranks to each SPE assigned to an application. By means of using PPE (and possibly conventional CPUs like Opterons if they exist in the cluster) primarily for shuttling messages to SPEs in other blades (or PS3s) instead of computation, the abstraction provided by CML allows each SPE to communicate with other SPEs regardless of whether the SPEs are in the same socket, the same blade, the same node, or different nodes. On a cluster of Cells, CML implements a mechanism for forwarding data from a SPE to its PPE then across a network to a remote PPE and finally to the target SPE. The PPE needs to be involved because a SPE cannot interact directly with I/O-bus devices such as network interface cards (NICs). In addition to handling communication operations, PPE, also initializes CML, starts SPE programs and waits until all SPEs invoke *MPI_Finalize*(), and finally shuts down the CML. Therefore both SPE/PPE programs need definitions of CML functions and should be linked with CML libraries, whereas SPE program can run an existing MPI application with only minor modifications that are necessary due to architectural requirements of the Cell. In effect, we have ported our scientific applications relatively easily to both of our testing platforms.

CML also provides Programmer's Message Passing Interface (PMPI) functions [12] which have a one-to-one correspondence to MPI calls. This interface enables any calls made to the MPI functions, by the SPEs, to be intercepted and thus recorded. Section 3.2 discusses the use of PMPI calls within our profiler.

CML also offers a remote procedure call (RPC) mechanism through which SPEs can invoke a function on the PPE (PPEs can subsequently call a function on the accompanying host CPU if it exists) and receive any results. This capability is particularly useful for our profiler, where local SPEs need to call a PPE *malloc()* to allocate space in PPE memory to hold the entire list of recorded events.

3. Software Design Details

Our implementation of the tracing library targets clusters of Cell processors. Each PPE within a Cell processor is responsible for synchronizing the program run on its SPEs. CML enables the total number of SPEs, as seen by an application, to scale: from a single processor containing eight SPEs to clusters of PS3s [10], or to Roadrunner that contains 97,920 SPE cores. The remainder of this section outlines the design and implementation of the profiler including its memory use, and events that are profiled.

3.1. *Data Structures*

The buffers that are used in the profiler implementation, along with the double-buffering operation of the buffers in the LS, is shown in Fig. 1. This is discussed further in Section 3.2.

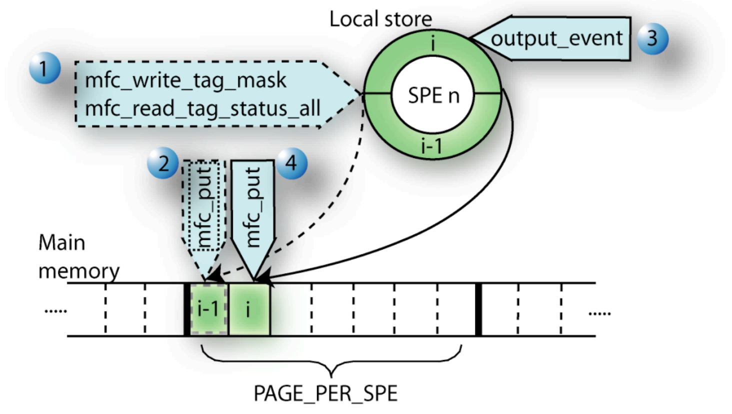


Fig. 1. Operation of the profiler double-buffering implementation.

A cyclical pattern is used in Fig. 1 to illustrate the allocation of buffers in LS. They switch roles repeatedly—while one is being used to record newly created events, the other is being dumped to PPE main memory. In comparison, the PPE memory layout is linear, where each small section, or event page, corresponds to the size of a single buffer in LS.

Table 1 summarizes the structure of profile events and of event pages that hold a number of events. It is crucial that the events and the buffers are allocated to fit the 16-byte boundary required for DMA transactions. *ALIGNED16* is a short-hand notation for the *__attribute__((aligned (16)))* attribute, which specifies to the compiler to allocate the data structure to be 16 byte aligned. It is also important that 16 byte aligned profile data is structured the same both on SPE and PPE memories.

The enumerator lists the type of events our implementation can currently monitor. We record calls to the profiler start/stop functions, SPE function entry/exit (*E, X*), calls to the MPI functions implemented in CML (*MPI_SEND, MPI_RECV, MPI_ALLREDUCE, MPI_REDUCE, MPI_BARRIER, MPI_BCAST*) and various DMA put/get transactions which are issued by functions *spu_mfcdma32*() and *spu_mfcdma64*() defined in *libspe2*—the standard SPE library included in the IBM Cell/B.E. SDK. We have limited our implementation to cover only relevant DMA transaction types to our test applications.

Table 1. Definitions of the data structures.

```
#define PAGE_SIZE 64
#define ALIGNED16 __attribute__((aligned
(16)))
```

Table 1. Cont'd.

```
  typedef enum      { PROFILE_START,
PROFILE_STOP, E, X, MPI_SEND, MPI_RECV,
MPI_ALLREDUCE, MPI_REDUCE, MPI_BARRIER,
MPI_BCAST, MFC_PUT, MFC_PUT64, MFC_GET64,
MFC_PUT32, MFC_GET32} event_type_t;

  typedef struct    {
     double time_stamp;
      double duration;
      event_type_t type;
      unsigned long long enx;
      unsigned long long exx;
      short output_flag;
      int data[8] ALIGNED16;
             }ALIGNED16 event_record_t;

  typedef struct page_tag {
      struct page_tag* next_page;
      event_record_t events[PAGE_SIZE];
             }ALIGNED16 event_page_t;
```

In addition to recording the type of event, *event_record_t* also records a time stamp and the duration of an event, address of the called SPE function and its caller (*enx ,exx*), an output flag to indicate that an event has happened and *data* array which includes destination/source, send/receive size and send/receive counts for MPI events. Effective addresses (*enx, exx*) are stored as an *unsigned long long* on both the SPE and PPE, so that they can be treated in a unified fashion no matter if the PPE code is compiled for 32-bit or 64-bit execution. One event record uses 80 bytes in memory.

A single buffer, or event page, is defined by *event_page_t*. The size of a page was set to be 64 in our testing (using 5,120 bytes). A pointer to the next page to use is a part of *event_page_t* in case the current page fills up. Contrary to the SPE, which has two event pages, the PPE allocates a far greater number of event pages. For our analysis on the PowerXCell 8i based hybrid cluster, the PPE allocates 10,000 event pages per SPE giving a total PPE memory footprint of 400 MiB (=8×10,000×64×80). However, in our PS3 benchmarks we had to limit the PPE memory allocation to less than 200 MB due to 20 times less PPE addressable memory in PS3. In fact, we have observed that as long as enough PPE memory is reserved, the performance of the profiler is not affected.

3.2. *Implementation*

CML based applications first start on the PPEs, which subsequently launch code on the SPEs. When the profiler is enabled, an instrumented SPE program, once launched, immediately invokes an allocation function on the PPE, using the CML's RPC mechanism, for event pages in main memory. Each SPE is returned the base address of the reserved memory via the same RPC mechanism. Before a SPE proceeds with actual application execution, it allocates two event buffers in its LS. However, this allocation is much smaller than its counterpart in main memory due to the limited size of the LS. In our tests the profiler statically allocates only two small event buffers of size 5,120 bytes, which holds up to 64 events, in SPE memory. Apart from the 10 KiB required for the two buffers, the profiler code requires less than an additional 2 KiB in LS but is dependent on the actual number of CML functions used by an application. This is ~30,000 times smaller than the memory used by our profiler in the main memory of the PPE of a PowerXCell 8i.

Profiler initialization is followed by the execution of the actual MPI application. Throughout the application run, the instrumented functions are called to record events. The instrumented operations, as provided by the profiler, create event logs. For instance, an SPE-to-SPE message-passing request invokes the corresponding instrumented MPI communication operation, which populates the event data structure with the relevant information, e.g., type, source/target, size of the message, and secondly calls the corresponding PMPI routine, which is implemented by CML, to send the actual message. The profiler library provides similar instrumented functions to profile other events including DMA operations and SPE function call activities.

SPE LS memory is limited to 256 KB. If it were to be filled with trace data, it would inhibit the execution of the SPE code. In order to circumvent this possibility, we use a double-buffering approach [13] to log trace events. Instead of continuously pushing events to a dynamically increasing allocation in LS, SPU writes profile event logs as they appear to one of the two small buffers allocated during profiler initialization. Once the buffer being used is full, previous buffer-dump operation is checked for completion (Step 1 in Fig. 1), by using *mfc_write_tag_mask* and *mfc_read_tag_status_al,* in order to avoid overwriting data being transferred. If the preceding dump has been completed, a non-blocking DMA (*mfc_put*) is issued to transfer the buffer to main memory (Step 2 in Fig. 1). Each SPE sends the data to a privately reserved address, which it determines by using the memory base address received through the RPC mechanism during initialization, its local rank and number of previous dumps it has performed up to then. The SPE also switches the trace buffers and uses the available buffer to record new events (Step 3 in Fig. 1). Meanwhile, the SPE execution continues without interruption as a non-blocking DMA is used. Once the second buffer is filled, the SPE switches buffers again and continues with recording events to the first buffer as it issues a DMA transfer (*mfc_put*) to dump the second buffer to the end of the preceding dump in the main memory (Step 4 in Fig. 1). If the speed of event generation is faster than the time taken to transfer a single LS buffer to main memory then the application execution will pause. In such a case the size of the LS buffers can be increased but clearly at a reduction in the size of the LS store available to the application.

The double-buffering implementation not only overlaps data dumping with program execution, but also gives the capability of logging in excess of 10^4 times more events than the LS could have stored by using just two small buffers, and leaves more LS available for program and data in each SPE.

Upon the termination of tracing, the SPE program dumps the last buffer, regardless of how full it is, to main memory. Once all of the SPEs terminate the PPE writes the profile data from main memory to several files, one per SPE, which contains the events that are ordered in terms of their time of occurrence. The output files can be post-processed for numerous performance analysis studies.

4. Results

In this section, we first provide detailed analysis of the overhead incurred by the profiling activity on a single Cell BE processor of the PS3. Secondly, we compare the overhead on a single PowerXCell 8i to that on Cell BE and finally delineate cluster-wide use of the profiler.

Three applications were chosen to both quantify the overheads of the profiler use and also to illustrate its usefulness. The first application is Sweep3D, which solves a single-group time-dependent discrete ordinates neutron-transport problem. It processes a regular three-dimensional data grid, which is partitioned onto a logical two-dimensional processor array. Its computation consists of a succession of 3D wave fronts (sweeps), in which each processor receives boundary data from upstream neighbors, performs a computation on its local sub-grid, and produces boundaries for downstream neighbors. All communications use MPI to transfer boundary data to neighboring processors.

The second application is molecular-dynamics (MD) [14]. The MD simulation follows the time evolution of the positions of *N* atoms by solving coupled ordinary differential equations. For parallelization, the MD code uses a 3-D spatial domain that is partitioned in all three dimensions into *P* sub-grids of equal volume. Each step in the simulation requires the processing of the local sub-grid as well as boundary exchanges in each of 6 neighboring directions (i.e. the lower and higher neighbor subsystems in the x, y and z directions).

The third application is a lattice Boltzmann (LB) method for fluid flow simulations. The cellular-automata like application represents fluid by a density function on of the grid points on a regular 3D lattice [10]. LB exhibits the same 3D communication pattern as for MD where each time step involves DF updates and inter-sub-grid density migrations.

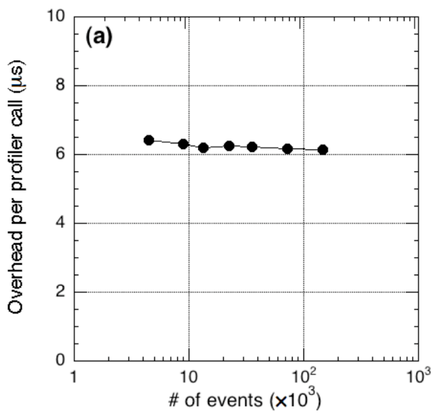
4.1. *Performance Overhead Analysis*

In effect, the performance overhead of the profiler is dependent on the application as the mixture of communication and computation operations vary from code to code. Therefore in this subsection we use an overhead metric by considering a worst-case scenario by using a kernel application containing only communication calls and no computation. Additionally, by executing the kernel on a single Cell processor we ensure that only fast on-chip communications over the EIB are used. The kernel application simply contains the communication pattern of the Sweep3D application thus resulting in a maximum rate of event generation. We provide the results first on Cell BE and second on PowerXCell 8i in the remainder of this subsection.

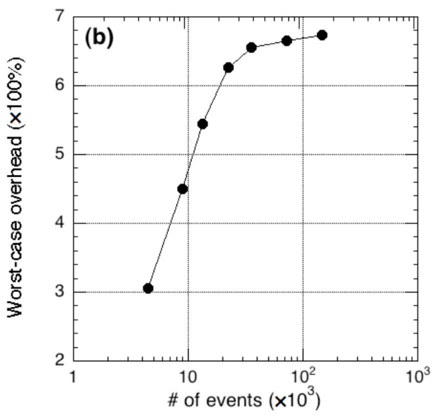
4.1.1. *Cell BE*

In order to quantify the profiling overhead we have performed a suite of tests on the Cell BE of the PS3, which represents typical node of our target cluster.

An equal number of MPI send and receive calls, using a fixed size of 600 doubles (4,800 bytes), for the 6 functional SPEs on the single Cell BE of the PS3 were used for the results shown in Fig. 2. Fig. 2(a) shows the average overhead for each profiler call as a function of the number of events and Fig. 2(b) shows the slowdown when varying the number of events (the x-axis shows the logarithm of the number of events). It can be seen that the average time required to record a single event is less than 6.3 μs. This corresponds to a slowdown of a factor of 6.8 for large numbers of events as shown in Fig. 2(b).



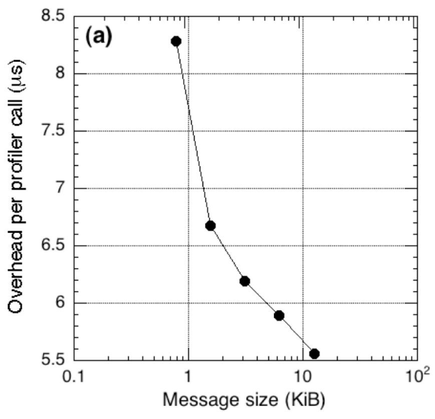
(a) Average cost per profiler call



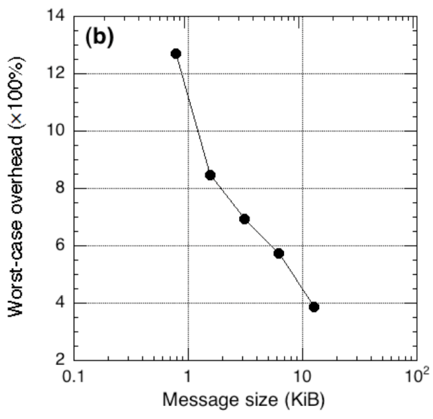
(b) Slowdown due to the profiler

Fig. 2. Performance overheads of the profiler (6 SPE run on a single Cell BE).

In order to quantify the effect of message size on profiling overhead, we have performed benchmarks on the Cell BE, and the results are shown in Fig. 3. Here, we fix the event count at 36,000, which is the point where saturation starts in Fig. 2(b), and keep the buffer size at 5 KiB while varying the sizes of the sent/received messages.



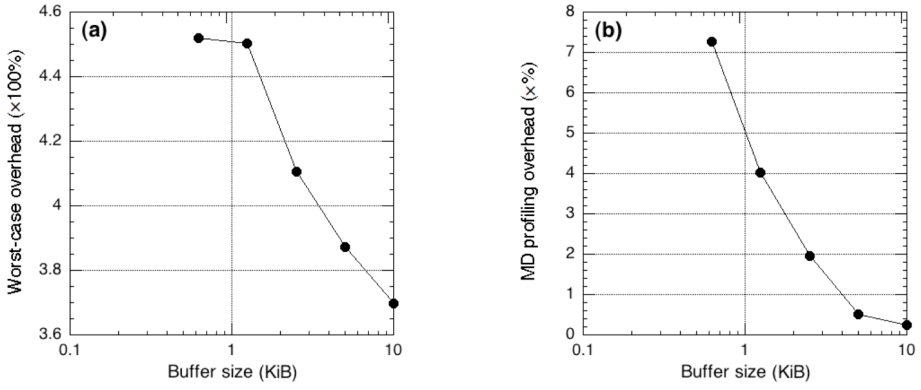
(a) Average cost per profiler call



(b) Slowdown due to the profiler

Fig. 3. Performance overheads of the profiler for varying message sizes (6 SPE run on a single Cell BE).

Fig. 3(a) shows that the smallest per profiler call overhead is less than 5.6 μs for 12.5 KiB sized messages, whereas Fig. 3(b) shows 4× slowdown factor. In comparison to Fig. 2, which used 4,800-bytes messages, profiler shows a better performance for 12.5 KiB messages. This can be attributed to the fact that larger transfers take longer time to complete, which is overlapped by keeping record of profile events, thereby reducing the profiler overhead.



(a) Slowdown in worst-case kernel (b) Slowdown in MD application

Fig. 4. Performance overhead of the profiler for varying SPE buffer sizes on the Cell BE.

In Fig. 4, we study the effect of changing the sizes of the double buffers reserved for profile events at SPEs of the Cell BE processor. In Fig. 4(a), we fix the event count at the saturation point of Fig. 2(b), i.e. 36,000, the message size at 12.5 KiB and vary the buffer size. It is observed that increasing the buffer size decreases the slowdown factor to as low as 3.7. As the buffer size at SPE increases, it takes less DMA transfers to PPE to dump the filled buffers, thereby increasing the performance of the profiler. However, it should be noted that there is a trade off between increasing the buffer size to achieve better profiler performance and the application performance itself because of the limited local store of SPEs. In selecting the buffer size, the memory requirements of application for its instructions and data should also be considered.

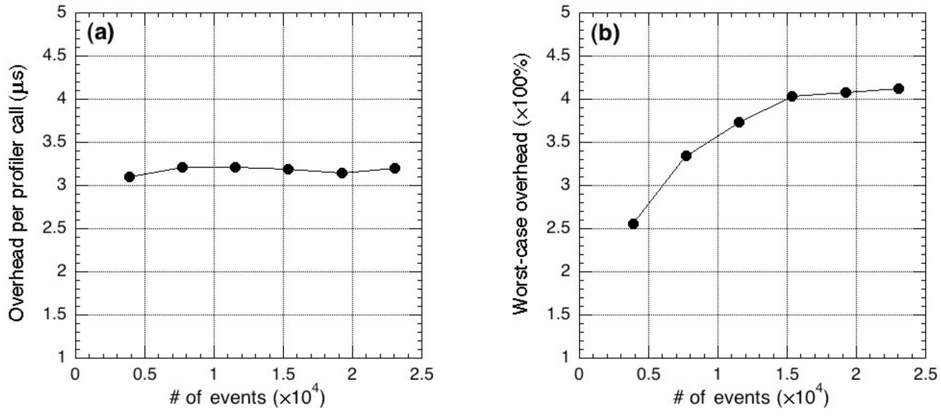
In order to quantify the profiling overhead for a typical scientific application, we port our parallel molecular-dynamics (MD) code [14] to PS3 using CML for handling MPI operations. Fig. 4(b) shows the overhead incurred by profiling of the message passing events of the MD application on a single PS3. The MD application implements a velocity-Verlet scheme, and calls several small functions at each time step for calculating atomic positions and velocities besides communication functions. Therefore, in Fig. 4(b), we turn off the function entry/exit tracing feature of the profiler in order to analyze MPI calls only, which are mainly for exchanging boundary-atom information. The results are the averages over a 100-time step simulation. Similar to Fig. 4(a), increasing the buffer size decreases profiling overhead. However, it should be noted that the y-axis of Fig. 4(b) is in percentages, i.e., profiler overhead is less than 0.5% for 10 KiB buffers. In effect, the cost of a single profiler call is less than 0.3 μs when 10 KiB buffers are used. The

MPI messages in the benchmark shown in Fig. 4(b) are on-chip communications and use the EIB of the Cell processor, which is much faster in comparison to current high performance computing interconnects. Therefore in a cluster-wide analysis of an MPI-centric scientific application, the overhead incurred by logging MPI events will be extremely low.

4.1.2. *PowerXCell 8i*

Here, we compare the overhead of the profiler on PowerXCell 8i to that on Cell BE before we proceed with PowerXCell 8i based cluster-wide experiments in the next subsection.

The overhead evaluation benchmark we described in the experiment of Fig. 2 is repeated on 8 SPEs of a single PowerXCell 8i processor, and the results are shown in Fig. 5. Fig. 5(a) shows that the average time required to record a single event is less than 3.2 μs and slowdown factor rises up to 4.2 for larger numbers of events as shown in Fig. 5(b). Recall however that intra-cell communications take full advantage of the EIB which has a total bandwidth of 204.8 GB/s. Intra-cell communications using CML actually achieve a bandwidth of ~23 GB/s and latency of ~0.3 μs as shown in Table 2. And hence a SPE-to-SPE message of size 4,800 bytes takes less than 1 μs within a single Cell.



(a) Average cost per profiler call (b) Slowdown due to the profiler

Fig. 5. Performance overheads of the profiler (8 SPE run on a single PowerXCell 8i).

It should be noted that the x-axis is linear in Fig. 5 as opposed to the logarithmic scale in Fig. 2. This is because the saturation of slowdown factor happens faster in comparison to Cell BE benchmark. The drop down in the average cost per profiler call and the slowdown factor in Fig. 5 in comparison to Fig. 2 can be explained by the fact that the Linux kernel in Cell BE of PS3 is running on top of a hypervisor which uses one SPE and also the EIB. On the PS3, the SPUs are hidden behind the hypervisor and every access happens in cooperation with the hypervisor. As a result writing to the local store or shuttling messages results in writing into kernel memory that represents the local store, which affects I/O of a given process and incurs additional overhead.

For a typical application running on a cluster of Cells, or on a hybrid processor configuration like Roadrunner, SPE-to-SPE communications can be significantly more costly than the ones considered in the worst-case discussed above. For instance, on Roadrunner, communications between SPEs on different nodes have a latency of over 11.7 μs at small message bandwidth of 161 MB/s. Therefore, in practice, the profiling overhead is much lower due to the increasing cost of communications as demonstrated in Fig. 4(b).

4.2. *Cluster-wide Profiling*

In this subsection we illustrate the usefulness of our profiling implementation for LB, MD and Sweep3D applications. We perform our tests on 8 nodes (32 SPEs per node) of a Roadrunner-like PowerXCell 8i based cluster.

4.2.1. *Communication analysis*

The information that is generated by the profiler is analyzed off-line. One log-file is generated for each SPE used by the application. Fig. 6 shows an example of the SPE-to-SPE communication pattern of the original Sweep3D code.

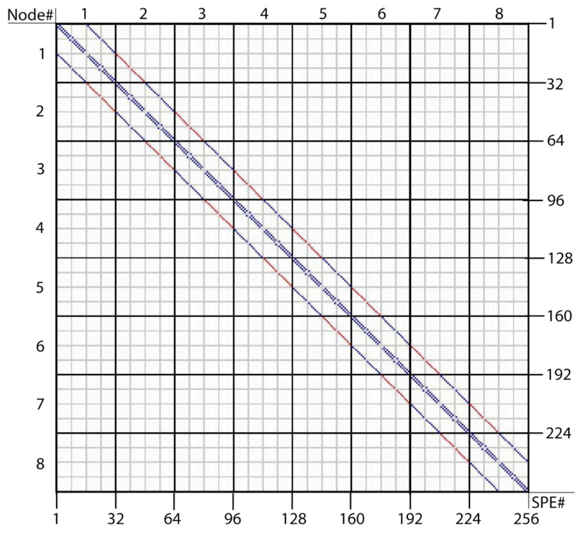


Fig. 6. Communication pattern of the original Sweep3D.

In Fig. 6, a larger square surrounded by thick lines and denoted by a node number, which contains 4×4 small squares, represents a tri-blade in the cluster. Each smaller square represents one Cell processor with 8 SPEs. The vertical and horizontal axes represent the sender and receiver SPE MPI ranks, and a colored pixel on the graph indicates a pair of communicating SPEs. The pixels are color coded to distinguish intra-node (blue) and inter-node (red) communications respectively.

The decomposition of Sweep3D's global grid onto a logical 2-D processor array can be seen in Fig. 6. Each processor communicates with its neighbor in the logical x and y directions. For a 256 processor run, the 2-D processor array consists of 16×16 SPE processors. Each processor communicates with its x neighbors (±1) as illustrated by the two sub-diagonals, and with its y neighbors (±16) indicated by the outermost two off-diagonals. Message passing for the two x neighbors is performed on the same chip through high-bandwidth (25.6 GB/s) EIB; whereas communication with the y neighbors corresponds to 1 message passing to another SPE residing within the same node but on the other Opteron and performed over PCIe via DaCS; and 1 message passing to an SPE on another node which adds InfiniBand in the path. These different inter-SPE communications incur different latency and bandwidth costs as shown in Table 2, which shows both the latency and bandwidth measured by ping-pong communication tests for CML.

Table 2. CML point-to-point performance.

Configuration	Latency	Bandwidth
Same Cell	0.272 μs	22,994.2 MB/s
Same node	0.825 μs	4,281.3 MB/s
Different nodes	11.771 μs	161.2 MB/s

The high latency of inter-node communication in comparison to intra-cell communication stems from the involvement of PPEs and Opterons in the former. To achieve higher performance, parallel algorithms should be designed to exploit the low latency and high bandwidth of EIB connecting intra-cell SPEs and avoid inter-node communication wherever possible. Fig. 7 shows the communication pattern of a modified version of Sweep3D, which performs much of the message passing activity over the EIB.

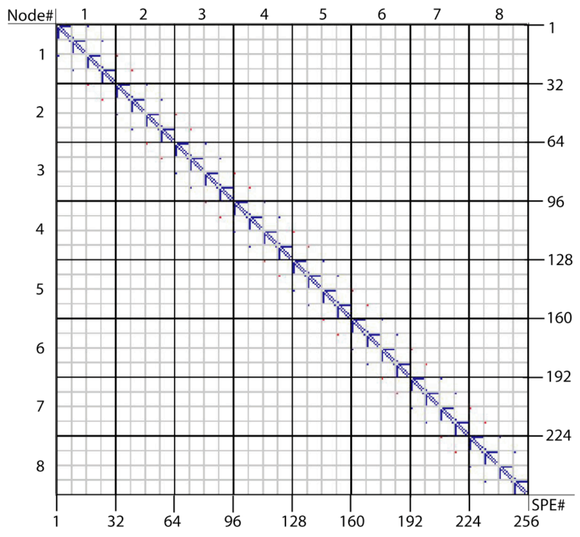


Fig. 7. Communication pattern of the modified Sweep3D.

In the modified version of Sweep3D, one SPE of each Cell acts as a root and exclusively handles inter-node message passing by gathering messages from the other SPEs on the same chip and sending it to the root on the destination Cell. This reduces the number of inter-cell messages significantly and promises an increase in performance.

Fig. 8 shows the communication pattern of MD for 256 SPE run. The logical arrangement of processors is in an 8×8×4 processor array. Each SPE performs two intra-cell communications with x neighbors. Communications to y neighbors is comparably slower with half of the SPEs requiring inter-node communications. For example, in the first node, SPEs 1–8 and SPEs 25–32 have one of their y neighbors in the next node, while for SPEs 9-24 the communications to y neighbors only involves intra-node communications. For all SPEs, message passing with z neighbors is inter-node communication with a high communication cost. This suggests a possible optimization, to increase the number of communications over the EIB, as with Sweep3D.

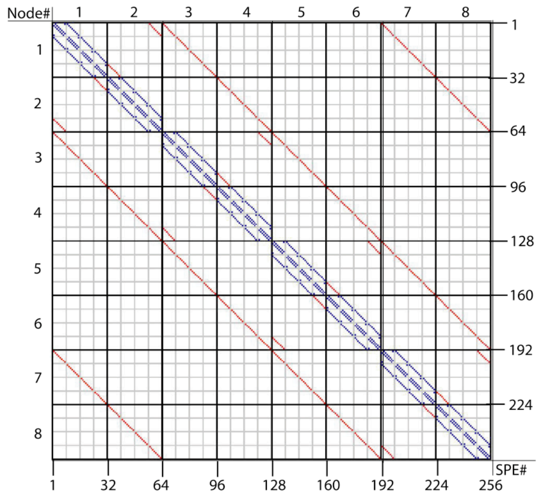


Fig. 8. Communication pattern of MD.

The volume of messages in MD is fairly regular. However, for applications where message send/receive activities and message sizes vary, heavier communication paths should be paid more attention. This is demonstrated with another application—LBM. LBM and MD have the same 3D communication pattern, however for LBM, some message passing events are an order-of-magnitude smaller than the others. Therefore, in Fig. 9, instead of plotting all communications, we have drawn only heavier communications.

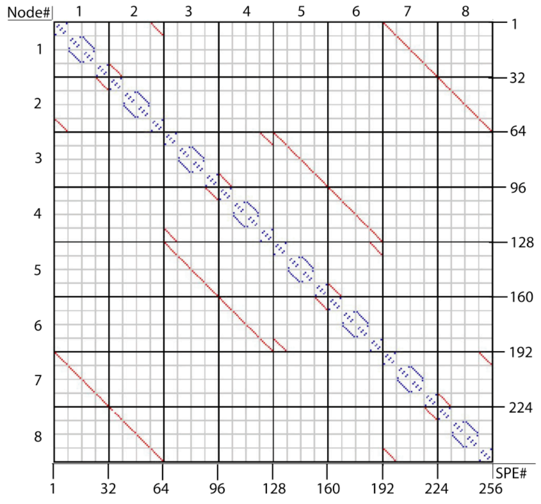


Fig. 9. Heavy communications of LB.

Fig. 9 has equal numbers of blue and red dots, indicating that 50% of message passing activity is inter-node. The other half of the communication is intra-node, of which 1/3 happens among SPEs on different Cells of the node. Therefore, only 1/3 of heavier communication is taking advantage of the fast EIB. This suggests that LBM suffers a larger communication cost in comparison with MD and that there is more room for optimization through the rearrangement of messages.

As a matter of fact, event data structure as described in Section 3.1 has enough data to provide finer details on message passing events. For example inter-SPE and/or SPE-to-PPE communications can be analyzed in finer detail. Function use, duration, type of message passing activity, size of the message, type of data being sent (and/or received), count of a certain data type, and source/destination, can be analyzed to provide more insight into the program flow. As we can extract point-to-point communication matrix from an application execution, it is also possible to automatically identify the communication pattern by measuring the degree of match between point-to-point communication matrix and predefined communication templates for regularly occurring communication patterns in scientific applications [15].

4.2.2. *Call-stack analysis*

The profiler library can also keep track of function entry and exits. This sub-subsection illustrates the use of this functionality by a call-stack analysis as another use of our profiler.

Fig. 10 shows the function call graph for the execution at the first SPE of a 256-SPE run for LB code. Instrumentation for 10 iterations is visualized and only a portion of call graph is provided for the clarity of presentation. The node shown as the root is the main function, which calls collision, streaming and communication functions once during every iteration. The nodes for these 3 functions include the source file name and the source code line information, which are looked up from a symbol table during post-processing. The node, which calls the *MPI_SEND/MPI_RECEIVE* implementation of CML, represent calls to the communication functions. Its children nodes show source/destination and data count of the message in parenthesis. The edges of the graph are marked with the number of times a particular event is observed. For instance, the node *MPI_SEND(0,56,132)* represents SPE 0 sends a message to SPE 56 of 132 bytes and it has occurred 10 times during the profiling.

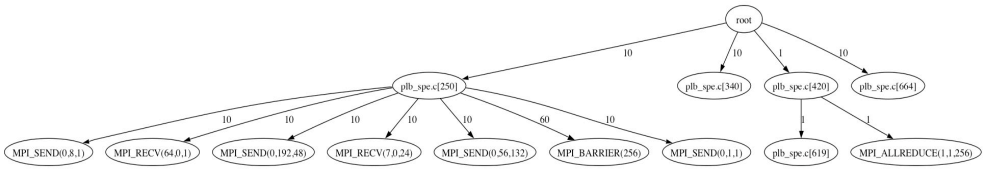


Fig. 10. Function call graph for LB.

The instrumentation is also done for functions expanded inline in other functions. The profiling calls indicate where the inline function is entered and exited. This requires that addressable versions of such functions must be available. A function may be given the attribute *no_instrument_function*, in which case this instrumentation will not be done. This can be used, for example, for high priority interrupt routines, and any function from which the profiling functions cannot safely be called, for example signal handlers.

The function call graph provides insight into program execution on a particular SPE on a cluster contributing to optimizations at the SPE level. In Fig. 10, we have weighted the edges with function call numbers. Instead, operation completion time could be used as an alternative for weighting as the profiler keeps durations of events as well. A call graph can be used to identify bottlenecks of performance at the SPE level and shed a light on required algorithm modifications for improvement.

5. Conclusions

We have developed a low-memory-footprint (12 KiB of local store), minimally intrusive profiling library for parallel applications running on clusters of Cell processors. Our library overlaps computations and DMA transfers to reduce application perturbation and efficiently utilizes the small amount of SPE local store available on Cell processors.

We have analyzed the performance of our profiler on the Cell BE processor of a PlayStation3 and explored profiler performance for varying design and application specific parameters, such as buffer and message size. We have used our profiler library to analyze the performance of parallel scientific applications that run across multiple Cell processors, Cell blades, and cluster nodes. Inter-blade communication analysis for Sweep3D has shown how communication structure can affect application performance.

We have ported two additional applications, LB and MD, to a hybrid Opteron+Cell cluster, and our profiler data suggests possible optimization opportunities. In order to demonstrate other uses of our library, we have analyzed the function-call pattern of a single SPE's program flow and used that to determine performance bottlenecks on the level of a SPE core.

While our study demonstrates high-speed, low-memory-overhead profiling for clusters augmented with Cell processors, it is certainly possible to optimize the profiler to further reduce its profiling cost and memory footprint. For example, the various types of profile events have different memory requirements (e.g., call-stack address records use only 16 bytes out of the 80 bytes allocated for the general event type). Therefore, restructuring the data types to be event-specific and adding compile-time options to customize the desired performance report may result in lower intrusion to program flow and reduce the post-processing effort for profile data.

Acknowledgements

This work was funded in part by the Advanced Simulation and Computing program of the Department of Energy, ARO, Chevron—CiSoft, DOE, DTRA, and NSF. Numerical tests were implemented using the AMD Opteron/IBM PowerXCell 8i cluster at the Los Alamos National Laboratory and Playstation3 cluster at the Collaboratory for Advanced Computing and Simulations, University of Southern California. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the US Department of Energy under contract DE-AC52-06NA25396.

References

[1] S. Pakin, Receiver-initiated Message Passing over RDMA Networks, in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, Miami, Florida, April 2008.

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, MPI: The Complete Reference, volume 1, The MPI Core. The MIT Press, Cambridge, Massachusetts, 2nd edition, September 1998.

[3] IBM, Software Development Kit for Multicore Acceleration Version 3.0: Programmer's Guide.

[4] H. Brunst and W. E. Nagel, Scalable Performance Analysis of Parallel Systems: Concepts and Experiences, in *Proceedings of the Parallel Computing Conference (ParCo 2003)*, Dresden, Germany, September 2003, pp. 737-744.

[5] D. Hackenberg, H. Brunst, and W. E. Nagel, Event Tracing and Visualization for Cell Broadband Engine Systems, in *Proceedings of 14th International Euro-Par Conference (Euro-Par 2008)*, Las Palmas de Gran Canaria, Spain, August 2008, pp. 172-181.

[6] Chen, R. Raghavan, J. N. Dale, and E. Iwata, Cell Broadband Engine Architecture and its First Implementation: A Performance View, *IBM Journal of Research and Development*, September 2007, 51(5):559-572.

[7] G. Michael, G. Fred, and F. P. Jan, High Performance Computing with the Cell Broadband Engine, in *Scientific Programming,* 2009, vol. 17, pp. 1-2.

[8] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, Entering the Petaflop Era: The Architecture and Performance and Roadrunner, in *Proceedings of IEEE/ACM SC08*, Austin, Texas, November 2008.

[9] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, Synergistic Processing in Cell's Multicore Architecture, *IEEE Micro*, March 2006, 26(2):10-24.

[10] K. Nomura, S. W. de Leeuw, R. K. Kalia, A. Nakano, L. Peng, R. Seymour, L. H. Yang, and P. Vashishta, Parallel Lattice Boltzmann Flow Simulation on a Low-cost Playstation 3 Cluster, *International Journal of Computer Science*, 2008.

[11] A. Buttari, J. Dongarra, and J. Kurzak, Limitations of the PlayStation 3 for high performance cluster computing, University of Tennessee Computer Science, Tech. rep. 2007.

[12] S. Mintchev, and V. Getov, PMPI: High-Level Message Passing in Fortran77 and C, in *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN 1997)*, volume 1225 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 603-614.

[13] J. C. Sancho, and D. J. Kerbyson, Analysis of Double Buffering on two Different Multicore Architectures: Quad-core Opteron and the Cell-BE, in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, Miami, Florida, April 2008.

[14] A. Nakano, R. K. Kalia, K. Nomura, A. Sharma, P. Vashishta, F. Shimojo, A. C. T. van Duin, W. A. Goddard, R. Biswas, D. Srivastava, and L. H. Yang, De Novo Ultrascale Atomistic Simulations On High-End Parallel Supercomputers, *International Journal of High Performance Computing Applications*, 2008, 22(1):113-128.

[15] D. J. Kerbyson, K. J. Barker, Automatic Identification of Application Communication Patterns via Template, *18th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS 2005)*, Las Vegas, Nevada, September 2005, pp. 114-121.