Dynamic Multigrain Parallelization on the Cell Broadband Engine

Filip Blagojevic,
Dimitrios S. Nikolopoulos
Department of Computer Science
Virginia Tech
660 McBryde Hall
Blacksburg, VA 24061
{filip,dsn}@cs.vt.edu

Alexandros Stamatakis
School of Computer &
Communication Sciences
Swiss Federal Institute of
Technology
Station 14, Ch-1015 Lausanne,
Switzerland
Alexandros.Stamatkis@epfl.ch

Christos D. Antonopoulos
Department of Computer Science
College of William and Mary
McGlothlin-Street Hall
Williamsburg, VA 23187-8795
cda@cs.wm.edu

ABSTRACT

This paper addresses the problem of orchestrating and scheduling parallelism at multiple levels of granularity on heterogeneous multicore processors. We present policies and mechanisms for adaptive exploitation and scheduling of multiple layers of parallelism on the Cell Broadband Engine. Our policies combine event-driven task scheduling with malleable loop-level parallelism, which is exposed from the runtime system whenever task-level parallelism leaves cores idle. We present a runtime system for scheduling applications with layered parallelism on Cell and investigate its potential with RAxML, a computational biology application which infers large phylogenetic trees, using the Maximum Likelihood (ML) method. Our experiments show that the Cell benefits significantly from dynamic parallelization methods, that selectively exploit the layers of parallelism in the system, in response to workload characteristics. Our runtime environment outperforms naive parallelization and scheduling based on MPI and Linux by up to a factor of 2.6. We are able to execute RAxML on one Cell four times faster than on a dual-processor system with Hyperthreaded Xeon processors, and 5–10% faster than on a single-processor system with a dual-core, quad-thread IBM Power5 processor.

1 INTRODUCTION

In the quest for delivering higher performance to scientific applications, hardware designers began to move away from conventional scalar processor models and embraced architectures with multiple processing cores. Although most commodity microprocessor vendors are already marketing multicore processors, these processors are largely based on replication of simple scalar cores. Unfortunately, these scalar designs exhibit well-known performance and power limitations. These limitations, in conjunction with a sustained requirement for higher performance, stimulated a renewed interest in unconventional processor designs. The Cell Broadband Engine (BE) is a representative of these designs, which has recently drawn considerable attention by industry and academia. Since it was originally designed for the game box market, Cell has low cost and a modest power budget. Nevertheless, the processor is able to achieve unprecedented peak performance for some real-world applications. IBM announced recently the use of Cell chips in a new Petaflop system with 16,000 Cells, due for delivery in 2008.

The potential of the Cell BE has been demonstrated convincingly in a number of studies [6, 19, 25]. Thanks to eight high-frequency execution cores with pipelined SIMD capabilities, and an aggressive data transfer architecture, Cell has a theoretical peak performance of over 200 Gflops for single-precision FP calculations and a peak memory bandwidth of over 25 Gigabytes/s. These performance figures position Cell ahead of the competition against the most powerful commodity microprocessors. Cell has already demonstrated impressive performance ratings in applications and computational kernels with highly vectorizable data parallelism, such as signal processing, compression, encryption, dense and sparse numerical kernels [6, 9, 20].

This paper explores Cell from a different perspective, namely that of multigrain parallelization. The Cell BE is quite unique as a processor, in that it can exploit orthogonal dimensions of task and data parallelism on a single chip. The processor is controlled by an SMT Power Processing Element (PPE), which usually serves as a scheduler for computations off-loaded to 8 Synergistic Processing Elements (SPEs). The SPEs are pipelined SIMD processors and provide the bulk of the Cell's computational power.

A programmer is faced with a seemingly vast number of options for parallelizing code on Cell. Functional and data decompositions of the program can be implemented on both the PPE and the SPEs. Typically, heavier load should be placed on the SPEs for higher performance. Functional decompositions can be achieved by splitting tasks between the PPE and the SPEs and by off-loading tasks from the PPE to the SPEs at runtime. Data decompositions may exploit the vector units of the SPEs, or be parallelized at two levels, using loop-level parallelization across SPEs and vectorization within SPEs. Functional and data decompositions can be static or dynamic, and they should be orchestrated to fully utilize both the eight SPEs and the PPE. Although the Cell vendors already provide programming support for using some of the aforementioned parallelization options, actually combining and scheduling layered parallelism on Cell can be an arduous task for the programmer.

To simplify programming and improve efficiency on Cell, we present a set of dynamic scheduling policies and the associated mechanisms. The purpose of these policies is to exploit the proper layers and degrees of parallelism from the application, in order to maximize efficiency of the Cell's computational cores. We explore the design and implementation of our scheduling policies using RAxML [24]. RAxML is a computational biology code, which computes large phylogenetic trees, using the Maximum Likelihood (ML) criterion. RAxML is extremely computationally intensive. The code is embarrassingly parallel at the task-level and exhibits intrinsic loop-level parallelism in each task. Therefore, it is a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

good candidate for parallelization on Cell. For a real world biological analysis, RAxML typically needs to execute 100 and 1000 tree searches as well as bootstrapped tree searches. Each of those searches represents an independent task, with single-nested loops that are both parallelizable and vectorizable.

This paper makes the following contributions:

- We present a runtime system and scheduling policies that exploit polymorphic (task and loop-level) parallelism on Cell. Our runtime system is adaptive, in the sense that it chooses the form and degree of parallelism to expose to the hardware, in response to workload characteristics. Since the right choice of form(s) and degree(s) of parallelism depends non-trivially on workload characteristics and user input, our runtime system unloads an important burden from the programmer.
- We show that dynamic multigrain parallelization is a necessary optimization for sustaining maximum performance on Cell, since no static parallelization scheme is able to achieve high SPE efficiency in all cases.
- We present an event-driven multithreading execution engine, which achieves higher efficiency on SPEs by oversubscribing the PPE.
- We present a feedback-guided scheduling policy for dynamically triggering and throttling loop-level parallelism across SPEs on Cell. We show that work-sharing of divisible tasks across SPEs should be used when the event-driven multithreading engine of the PPE leaves more than half of the SPEs idle. We observe benefits from loop-level parallelization of off-loaded tasks across SPEs. However, we also observe that loop-level parallelism should be exposed only in conjunction with low-degree task-level parallelism. Its effect diminishes as the degree of task-level parallelism in the application increases.

To put our study in a broader context, we present comparisons of the Cell BE against IBM Power5, a leading multicore processor with SMT cores, and against a dual-processor SMP system with Hyperthreaded Xeon processors. Cell outperforms both platforms. Taking into account cost and power efficiency, Cell exhibits great promise as the processor of choice for high-end systems and challenging applications.

The rest of this paper is organized as follows: Section 2 provides a brief overview of related work on Cell. Section 3 describes RAxML. Section 4 outlines the architecture of the Cell BE. Section 5 presents our runtime system and scheduling policies, along with their experimental evaluation. Section 6 summarizes the paper.

2 Related Work

We briefly summarize published research on Cell, which includes performance analysis of various aspects of the processor, and various compiler/runtime support environments.

Kistler et. al [15] present results on the performance of the Cell's on-chip interconnection network. They show a series of experiments that estimate the DMA latencies and bandwidth of Cell, using microbenchmarks. They also investigate the system behavior under different patterns of communication between local storage and main memory. Williams et. al [26] present an analytical framework to predict performance on Cell. In order to test their model, they use several computational kernels, including dense matrix multiplication, sparse matrix vector multiplication, stencil computations, and 1D/2D FFTs. In addition, they propose micro-architectural modifications that can increase the performance of Cell when operating on double-precision floating point elements. Our work considers the performance implications of multigrain parallelization strategies on Cell, using a real-world parallel application from the area of computational biology.

Eichenberger et. al [9] present several compiler techniques targeting automatic generation of highly optimized code for Cell. These techniques attempt to exploit two levels of parallelism, thread-level and SIMD-level, on the SPEs. The techniques include compiler assisted memory alignment, branch prediction, SIMD parallelization, OpenMP thread-level parallelization, and compiler-controlled software caching. The study of Eichenberger et. al. does not present details on how multiple levels of parallelism are exploited and scheduled simultaneously by the compiler. Scheduling layered and polymorphic parallelism is a central theme of this paper. The compiler techniques presented in [9] are also complementary to the work presented in this paper. They focus primarily on extracting high performance out of each individual SPE, whereas our work focuses on scheduling and orchestrating computation across SPEs.

Although Cell has been a focal point in numerous articles in popular press, published research using Cell for real-world HPC applications beyond games is scarce. Hjelte [14] presents an implementation of a smooth particle hydrodynamics simulation on Cell. This simulation requires good interactive performance, since it lies on the critical path of real-time applications such as interactive simulation of human organ tissue, body fluids, and vehicular traffic. Benthin et. al [3] present an implementation of ray-tracing algorithms on Cell, targeting also at high interactive performance.

3 RAxML-VI-HPC

RAxML-VI-HPC (v2.1.3) (Randomized Axelerated Maximum Likelihood version VI for High Performance Computing) [24] is a program for large-scale ML-based (Maximum Likelihood [11]) inference of phylogenetic (evolutionary) trees using multiple alignments of DNA or AA (Amino Acid) sequences. The program is freely available as open source code at icwww.epfl.ch/˜stamatak (software frame).

Phylogenetic trees are used to represent the evolutionary history of a set of n organisms. An alignment with the DNA or AA sequences representing those n organisms (also called taxa) can be used as input for the computation of phylogenetic trees. In a phylogeny the organisms of the input data set are located at the tips (leaves) of the tree whereas the inner nodes represent extinct common ancestors. The branches of the tree represent the time which was required for the mutation of one species into another, new one. The inference of phylogenies with computational methods has many important applications in medical and biological research (see [2] for a summary). An example for the evolutionary tree of the monkeys and the homo sapiens is provided in Figure 1.

Due to the rapid growth of sequence data over the last years, it has become feasible to compute large trees which often comprise more than 1,000 organisms and sequence data from several genes (so-called multi-gene alignments). This means that alignments grow in the number of organisms as well as in sequence length. The computation of the tree-of-life containing representatives of all living beings on earth is still one of the *grand challenges* in Bioinformatics.

The fundamental algorithmic problem computational phylogeny faces consists in the immense amount of alternative tree topologies which grows exponentially with the number of organisms n, e.g. for $n = 50$ organisms there exist $2.84 * 10^{76}$ alternative trees (number of atoms in the universe $\approx 10^{80}$). In fact, it has only recently been shown that the ML phylogeny problem is NP-hard [7]. In addition, ML-based inference of phylogenies is very memory- and floating point-intensive such that the application of high performance com-

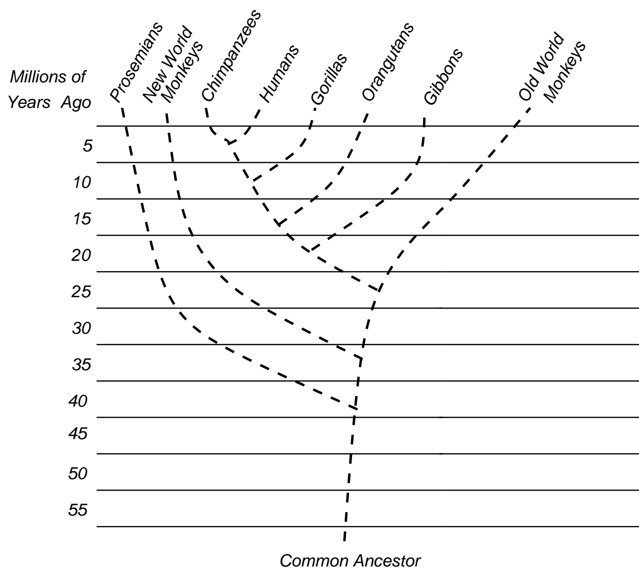

Figure 1. Phylogenetic tree representing the evolutionary relationship between monkeys and the homo sapiens

puting techniques as well as the assessment of new CPU architectures can contribute significantly to the reconstruction of larger and more accurate trees.

Nonetheless, over the last years there has been significant progress in the field of heuristic ML search algorithms with the release of programs such as IQPNNI [18], PHYML [13], GARLI [27] and RAxML [22, 24].

Some of the largest published ML-based biological analyses to date have been conducted with RAxML [12, 16, 17, 21]. The program is also part of the greengenes project [8] (greengenes.lbl.gov) as well as the CIPRES (CyberInfrastructure for Phylogenetic RESearch, www.phylo.org) project. To the best of the authors knowledge RAxML-VI-HPC has been used to compute trees on the two largest data matrices analyzed under ML to date: a 25,057-taxon alignment of protobacteria (length: 1,463 nucleotides) and a 2,182-taxon alignment of mammals (length: 51,089 nucleotides).

The current version of RAxML incorporates a significantly improved rapid hill climbing search algorithm. A recent performance study [24] on real world datasets with $\geq$ 1,000 sequences reveals that it is able to find better trees in less time and with lower memory consumption than other current ML programs (IQPNNI, PHYML, GARLI). Moreover, RAxML-VI-HPC has been parallelized with MPI (Message Passing Interface), to enable embarrassingly parallel non-parametric bootstrapping and multiple inferences on distinct starting trees in order to search for the best-known ML tree (see Section 3.1 for details). In addition, it has been parallelized with OpenMP [23]. Like every ML-based program, RAxML exhibits a source of fine-grained loop-level parallelism in the likelihood functions which consume over 90% of the overall computation time. This source of parallelism scales particularly well on large memory-intensive multi-gene alignments due to increased cache efficiency. Finally, RAxML has also recently been ported to a GPU (Graphics Processing Unit) [5].

3.1 The MPI Version of RAxML

The MPI version of RAxML exploits the embarrassing parallelism that is inherent to every real-world phylogenetic analysis. In order to conduct a “publishable” tree reconstruction a certain number (typically 20–200) of distinct inferences (tree searches) on the original alignment as well as a large number (typically 100-1,000) of bootstrap analyses have to be conducted (see [12] for an example of a real-world analysis with RAxML). Thus, if the dataset is not extremely large, this represents the most reasonable approach to exploit HPC platforms from a user's perspective.

Multiple Inferences on the original alignment are required in order to determine the best-known (best-scoring) ML tree (we use the term best-known because the problem is NP-hard). This is the tree which will then be visualized and published. In the case of RAxML, each independent tree search starts from a distinct starting tree. This means, that the vast topological search space is traversed from a different starting point every time and will yield final trees with different likelihood scores. For details on the RAxML search algorithm and the generation of starting trees, the reader is referred to [22].

Bootstrap Analyses are required to assign confidence values ranging between 0.0 and 1.0 to the internal branches of the best-known ML tree. This allows to determine how well-supported certain parts of the tree are and is important for the biological conclusions drawn from it. Bootstrapping is essentially very similar to multiple inferences. The only difference is that inferences are conducted on a randomly re-sampled alignment for every bootstrap run, i.e. a certain amount of columns (typically 10–20%) is re–weighted. This is performed in order to assess the topological stability of the tree under slight alterations of the input data. For a typical biological analysis, a minimum of 100 bootstrap runs is required.

All those individual tree searches be it bootstrap or multiple inferences are completely independent from each other and can thus be exploited by a simple master-worker scheme.

4 The Cell Broadband Engine

The main components of the Cell BE are a single Power Processing element (PPE) and eight Synergistic Processing Elements (SPEs) [10]. These elements are connected with an on-chip Element Interconnect Bus (EIB).

The PPE is a 64-bit, dual-thread PowerPC processor, with Vector/SIMD Multimedia extensions [1] and two levels of on-chip cache. The L1-I and L1-D caches have a capacity of 32 KB, while the L2 cache has a capacity of 512 KB. In this work we use a Cell blade with two Cell BEs running at 3.2 GHz, and 1GB of XDR RAM (512 MB per processor). The PPEs run Linux Fedora Core 5. We use the Toolchain 4.0.2 compilers.

The SPEs are the primary computing engines of the Cell processor. Each SPE is a 128-bit processor with two major components: a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). All instructions are executed on the SPU. The SPU includes 128 registers, each 128 bits wide, and 256 KB of software-controlled local storage. The SPU can fetch instructions and data only from its local storage and can write data only to its local storage. The SPU implements a Cell-specific set of SIMD instructions. All single precision floating point operations on the SPU are fully pipelined, and the SPU can issue one single-precision floating point operation per cycle. Double precision floating point operations are partially pipelined and two double-precision floating point operations can be issued every six cycles. Double-precision FP performance is therefore significantly lower than single-precision FP performance. With all eight SPUs active and fully pipelined double precision FP operation, the Cell BE is capable of a peak performance of 21.03 Gflops. In single-precision FP operation, the Cell BE is capable of a peak performance of 230.4 Gflops [6].

The SPE can access RAM through direct memory access (DMA) requests. The DMA transfers are handled by the MFC. All programs running on an SPE use the MFC to move data and instructions between local storage and main memory. Data transferred be-

tween local storage and main memory must be 128-bit aligned. The size of each DMA transfer can be at most 16 KB. DMA-lists can be used for transferring large amount of data (more than 16 KB). A list can have up to 2,048 DMA requests, each for up to 16 KB. The MFC supports only DMA transfer sizes that are 1,2,4,8 or multiples of 16 bytes long.

The EIB is an on-chip coherent bus that handles communication between the PPE, SPE, main memory, and I/O devices. The EIB is a 4-ring structure, and can transmit 96 bytes per cycle, for a bandwidth of 204.8 Gigabytes/second. The EIB can support more then 100 outstanding DMA requests.

5 Scheduling and Runtime Support for Multigrain Parallelization on Cell

In this section we present our scheduling policies and runtime support environment on Cell. We use RAxML to evaluate the policies and mechanisms. We first discuss briefly the optimization of RAxML bootstraps for the Cell SPEs (Section 5.1). We discuss our event-driven multithreading scheduler for task-level parallelization in Section 5.2. In Section 5.3, we discuss our adaptive loop scheduler and its implementation. Section 5.4 presents policies and mechanisms to adaptively merge task-level and loop-level parallelism.

5.1 SPE Optimizations

A straightforward adaptation of the MPI version of RAxML on Cell is to execute multiple MPI processes on the PPE and to have the major computational kernels of each process offloaded to an SPE. Each MPI process executes one RAxML bootstrap at a time. To identify the parts of RAxML that are suitable for SPE execution, we profiled the code executed by an MPI process using the `gprof` profiler. For all profiling and production run experiments presented in this paper, we used the file 42_SC as input to RAxML. 42_SC contains 42 organisms. Each organism is represented by a DNA sequence of 1167 nucleotides.

On an IBM Power processor, 98.77% of the execution time of RAxML is spent in the three main functions which compute the likelihood: 76.8% of execution time is spent in `newview()`, 2.37% of execution time is spent in `evaluate()`, and 19.6% of execution time is spent in `makenewz()`. These functions are the obvious candidates for off-loading to SPEs. We off-load the functions as a group, in a single code module loaded on each SPE. The advantage of having a single module is that it can be loaded to the local storage of an SPE once and reused throughout the execution of the application, unless a change in the degree or form of parallelism executed on the SPEs is dictated by the runtime system.

A possible drawback with merging off-loaded functions is that the larger size of the code module reduces the space available on the SPE for the stack and heap segments. In the case of RAxML, when all three functions are off-loaded, the total size of the code segment in the off-loaded file is 117KB. The remaining space in the local storage (139 KB) is sufficient for the stack and heap segments, since the working set of the SPE functions is small for realistic problem sizes. In the general case, off-loading should be controlled dynamically to achieve a good trade-off between code locality, data locality and overall performance.

Naive off-loading has negative effect on performance for RAxML. We measure the execution time of RAxML before and after the three dominant functions are off-loaded, using one thread on the PPE and one SPE. The execution time of RAxML before off-loading any function to an SPE with the 42_SC input is 38.23s. The execution time after off-loading the three functions increases to 50.38s. There are several reasons which explain the performance degradation caused by naive off-loading:

- The off-loaded code is working on double-precision floating point numbers, and the double-precision FP operations are neither vectorized, nor fully pipelined in the original code.
- Each mispredicted branch executed on an SPE incurs a 20 cycle penalty. In the off-loaded code, 45% of the execution time is spent in condition checking, and the inherently random distribution of branch targets in the code makes the outcome of the conditions hard to predict.
- The DMA transfers between the local storage and the main memory are not optimized.
- The code uses expensive mathematical functions such as `log()` and `exp()`.
- The communication between the PPE and the SPEs is not optimized.

We used this itemized list as a guideline for optimizing the off-loaded code of RAxML on Cell. We implemented vectorization of the ML calculation loops and vectorization of conditionals. We pipelined the vector operations, aggregated data transfers and replaced the original mathematical functions with numerical approximations of the same functions from the Cell SDK library. The specifics of these optimizations are beyond the scope of this paper. A detailed description is provided elsewhere [4]. The execution time of the optimized SPE code of RAxML was reduced from 50.38s to 28.82s, which corresponds to a speedup of 1.32 over single-threaded execution on the PPE. The optimizations apply 1 to 1 to multiple inferences on the original alignment.

5.2 Scheduling Task-Level Parallelism

Mapping MPI code on Cell can be achieved by assigning one MPI process to each thread of the PPE. Given that the PPE is a dual-thread engine, MPI processes on the PPE can utilize two out of the eight SPEs via concurrent function off-loading. We consider two strategies to use the rest of the SPEs on Cell. The first is multi-level parallelization, and more specifically loop-level parallelization within the off-loaded functions and loop distribution across SPEs. The second is a model for event-driven task-level parallelism, in which the PPE scheduler oversubscribes the PPE with more than two MPI processes, to increase the availability of tasks for SPEs.

We first examine the event-driven task parallelization model, since it provides an opportunity for coarse-grained parallelization. We will refer to this model as EDTLP, for event-drivel task-level parallelism, in the rest of the paper. In EDTLP, tasks correspond to off-loadable functions from MPI processes running concurrently or in a time-shared manner on the PPE. These tasks are served by running MPI processes using a fair sharing algorithm, such as round-robin. The scheduler off-loads a task immediately upon request from an MPI process, and switches to another MPI process while off-loading takes place. Switching upon off-loading prevents MPI processes from blocking the code while waiting for their tasks to complete on the SPEs.

EDTLP can be implemented using a user-level scheduler to interleave off-loading across MPI processes. The scheduler is simple to implement and it can be integrated transparently in the original MPI code, provided that the tasks that need to be off-loaded are annotated. EDTLP overcomes the problem of crippling underutilization of the cores, both PPEs and SPEs, when PPE threads do not have tasks to off-load, or when PPE threads wait for completion of already off-loaded tasks.

Multiplexing more than two MPI processes on the PPE intro-

duces system overhead due to context switching and due to implicit costs following context-switching across address spaces, such as cache and TLB pollution. Furthermore, the granularity of the off-loaded code is critical as to whether the multiplexing cost can be tolerated or not. The off-loaded code should be coarse enough to mask the overhead of multiplexing. Our EDTLP scheduler uses granularity control and voluntary context switches to address these issues.

Formally, the EDTLP scheduler executes a task graph comprising PPE tasks and SPE tasks. The scheduler follows a dependence-driven execution model. If the scheduler locates a task to be off-loaded in an MPI process, it searches for an idle SPE, and if it finds one, it sends a signal to begin task execution. If no idle SPE is found, the scheduler waits until an SPE becomes available. Let t_{spe} denote the execution time of a task on an SPE, t_{code} denote the time needed to load the code of a task on the SPE, and t_{comm} denote the time to send a signal from the PPE to an SPE to commence execution of an off-loaded task, or vice versa, to send a result from an SPE back to the PPE. The scheduler selects to off-load tasks that meet the condition $t_{spe} + t_{code} + 2t_{comm} < t_{ppe}$, where t_{ppe} is the execution time of the task on the PPE. Note that $t_{code} = 0$ if a task is executed on an SPE more than once and for all executions of the task other than the first. Our runtime system preloads annotated SPE functions to amortize the code shipping cost. The code remains on the SPEs, unless the runtime system decides to change its parallelization strategy and either trigger or throttle loop-level parallelism. This issue is discussed further in Section 5.4.

Since the scheduler does not know the length of the tasks a-priori, it optimistically off-loads any user-designated task and throttles off-loading for tasks that do not pass the granularity test. To implement this dynamic scheme, the code needs to maintain PPE and SPE versions of off-loadable functions. This is an easy modification, since PPE implementations of all off-loadable functions are available in the original MPI version of the code. The modification comes at the expense of an increased code footprint on the PPE. If the scheduler does not find tasks to off-load, it blocks until a new off-loading request originates from a PPE thread.

Figure 2 illustrates an example of the functionality of the EDTLP scheduler. The example uses PPE and SPE task sizes which are representative of RAxML functions. We show the execution of two off-loaded tasks, with an approximately 1:3 length ratio. In case (a), once a task is off-loaded by an MPI thread, the PPE switches context. At any time, two MPI threads can off-load tasks concurrently, however multiplexing the MPI threads with EDTLP enables the scheduler to use all 8 SPEs for a significant part of the coarse-grained function, and at least 4 SPEs for a significant part of the fine-grained function. In case (b), the scheduler runs persistently one MPI thread on the PPE until all functions from that task are off-loaded. The implication is that 6 out of the 8 SPEs remain idle most of the time. In RAxML, the off-loaded tasks have durations up to 96 μs. Their granularity is an order of magnitude finer than the granularity of the Linux scheduler's time quantum, which is a multiple of 10 ms. Therefore, the OS scheduler is highly unlikely to switch context upon function off-loading. The EDTLP scheduler resolves this problem, thus achieving higher SPE utilization. The context switching overhead on the PPE is 1.5 μs per switch. This overhead is low enough to tolerate up to 7 context switches while one RAxML task is running.

We evaluate our EDTLP scheduler by comparing its performance to the performance achieved with the Linux 2.6.17 scheduler and without user-level scheduling support on our Cell blade. In this evaluation, we use the fully optimized version of RAxML, outlined in Section 5.1. This version off-loads the three ML calculation functions to SPEs. From the total execution time of one non-parametric bootstrap analysis of RAxML, 90% is spent to compute

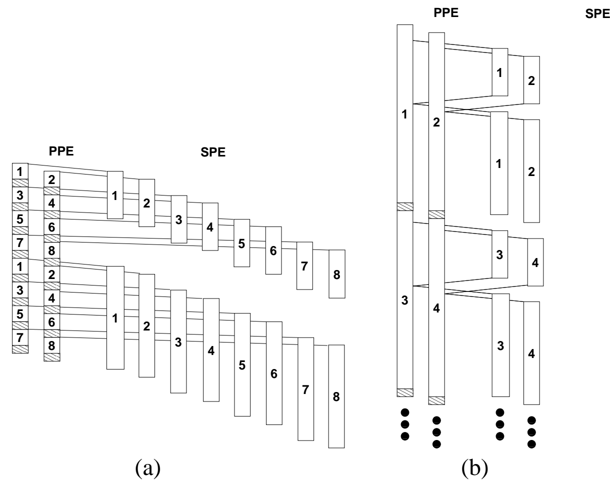

Figure 2. Scheduler behavior for two off-loaded tasks, representative of RAxML. Case (a) illustrates the behavior of our EDTLP scheduler. Case (b) illustrates the behavior of the Linux scheduler with the same workload. The numbers in the tasks correspond to MPI processes. The shaded slots indicate context switches.

	EDTLP	Linux
1 worker, 1 bootstrap	28.46s	28.42s
2 workers, 2 bootstraps	29.36s	29.23s
3 workers, 3 bootstraps	32.54s	56.95s
4 workers, 4 bootstraps	33.12s	57.38s
5 workers, 5 bootstraps	37.27s	85.88s
6 workers, 6 bootstraps	38.66s	86.43s
7 workers, 7 bootstraps	41.87s	114.92s
8 workers, 8 bootstraps	43.32s	115.51s

Table 1. Performance comparison for RAxML with different scheduling policies. The second column shows the execution times with EDTLP. The third column shows execution times with the Linux kernel scheduler. The input file is 42_SC.

on SPEs and 10% is spent to compute and schedule tasks on the PPE. The average SPE computing time is 96μs. The average PPE computing time between consecutive offloads is 11μs.

Table 1 summarizes the results obtained from the experiment. The first column shows the number of workers used, and the amount of work performed. RAxML is always executed in a massively parallel setting, with constant problem size (one bootstrap) per MPI process. The second column shows execution times of RAxML, when the MPI processes on the PPEs are scheduled with the EDTLP scheduler. The third column shows execution times when the MPI processes are scheduled by the Linux scheduler. Ideally, the total execution time should remain constant. The reasons why this is not the case consists in the sub-optimal (90%) coverage of parallel code executed on the SPEs, contention between MPI processes sharing the SMT pipeline of the PPE, and SPE parallelization and synchronization overhead. The EDTLP scheduler keeps the execution time within a factor of 1.5 of the optimal and achieves about 2.6 times the performance of the Linux scheduler.

5.3 Scheduling Loop-Level Parallelism

The EDTLP model described in Section 5.2 is effective if the PPE has enough coarse-grained functions to off-load to SPEs. In cases where the degree of available task parallelism is less than the number of SPEs, the runtime system can activate a second layer

of parallelism, by splitting an already off-loaded task across multiple SPEs. We implemented runtime support for parallelization of for-loops enclosed within off-loaded SPE functions. We parallelize loops in off-loaded functions using work-sharing constructs similar to those found in OpenMP. In RAxML, all for-loops in the three off-loaded functions have no loop-carried dependencies, and all loops obtain speedup from parallelization, assuming that there are enough idle SPEs dedicated to their execution. The number of SPEs activated for work-sharing is user-controlled or system-controlled, as in OpenMP. We discuss dynamic system-level control of loop parallelism further in Section 5.4.

As an example of loop-level parallelization, we use a loop from function `evaluate()`, shown in Figure 3.

```
for( i=... )
{
  term =  x1[i].a * x2[i].a;
  term += x1[i].c * x2[i].c * diagptable[i * 3];
  term += x1[i].g * x2[i].g * diagptable[i * 3 + 1];
  term += x1[i].t * x2[i].t * diagptable[i * 3 + 2];

  term = log(term) + (x2[i].exp) * log(minlikelihood);

  sum += wptr[i] * term;
}
```

Figure 3. A parallel loop in function `evaluate()` of RAxML.

The basic work-sharing scheme we used is presented in Figure 4. Before the loop is executed, a designated master SPE thread sends a signal to all SPE worker threads. After sending the signal, the main thread executes its assigned portion of the loop. The main thread and all the workers fetch the chunk of data they need to execute their portions of the loop from the shared RAM. Global shared data modified during loop executions are committed to RAM. Data needed by the master SPE upon loop completion to make forward progress, are sent directly from the worker SPEs via SPE to SPE communication, in order to avoid the latency of going through shared memory. SPE to SPE communication enables dependence-driven execution of multiple parallel loops across SPEs.

In the example in Figure 3, the SPEs perform first a local reduction. The master SPE accumulates the local sum received from worker SPEs in local storage and proceeds with execution after each worker SPE signals completion of the loop.

5.3.1 SPE-SPE communication

The SPE threads participating in loop work-sharing constructs are created once upon function off-loading. Communication among SPEs participating in work-sharing constructs is implemented using DMA transfers and the communication structure `Pass`, shown in Figure 5.

The `Pass` structure is private to each thread. The master SPE thread has an array of `Pass` structures. Each member of this array is used for communication with one SPE worker thread. Once the SPE threads are created, they exchange the local addresses of their `Pass` structures. This address exchange is done through the PPE. Whenever one thread needs to send a signal to a thread on another SPE, it issues an `mfc_put()` request and sets the destination address to be the address of the `Pass` structure of the recipient.

In Figure 6, we illustrate the loop from Figure 3, parallelized with work-sharing among SPE threads. Before executing the loop, the master thread sets the parameters of the `Pass` structure for each worker SPE and issues one `mfc_put()` request per worker. This is done in `send_to_spe()`. Worker i uses the parameters of the received `Pass` structure and fetches the data needed for the loop execution to its local storage (function `fetch_data()`). After finishing the execution of its portion of the loop, a worker sets locally the `res` parameter in the structure `Pass` and sends the structure `Pass` to the master, using `send_to_spe()`. The master merges the result from workers and commits it to main memory.

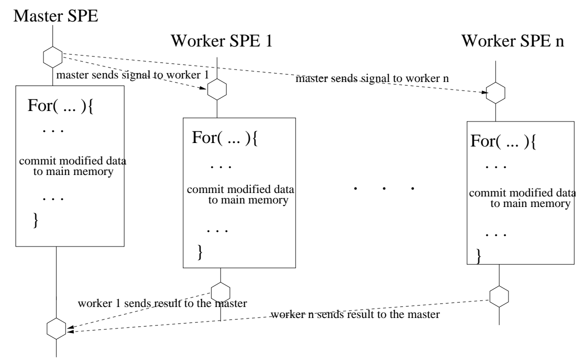

Figure 4. Parallelizing a loop across SPEs using a work-sharing model.

```
struct Pass{

    volatile unsigned int v1_ad;
    volatile unsigned int v2_ad;
    //...arguments for loop body
    volatile unsigned int vn_ad;
    volatile double res;
    volatile int sig[2];

} __attribute__((aligned(128)));
```

Figure 5. The data structure `Pass` is used for communication among SPEs. The *vi_ad* variables are used to pass the addresses of input arguments for the loop body from one local storage to another. The variable `sig` is used as a notification signal that the memory transfer for the shared data updated during the loop has completed. The variable `res` is used to send results back to the master SPE and as a dependence resolution mechanism.

Immediately after calling `send_to_spe()`, the master participates in the execution of the loop. The master tends to have a slight head start over the workers. The workers need to complete several DMA requests before they can start executing the loop, in order to fetch the required data from the master's local storage or shared memory. In fine-grained off-loaded functions such as those encountered in RAxML, load imbalance between the master and the workers is noticeable. To achieve better load balancing, we set the master to execute a slightly larger portion of the loop. A fully automated and adaptive implementation of this purposeful load unbalancing is obtained by timing idle periods in the SPEs across multiple invocations of the same loop. The collected times are used for tuning iteration distribution in each invocation, in order to reduce idle time on SPEs.

Table 2 shows the execution times of RAxML with one layer of loop-level parallelism exploited in the off-loaded functions. We execute one bootstrap of RAxML, to isolate the impact of loop-level parallelism. The number of iterations in each parallelized loop depends on the alignment length. For the 42_SC input file, the number of iterations in each parallelized loop is 228.

The results in Table 2 suggest that using up to five SPEs for loop parallelization achieves speedup over loop execution using one SPE. The maximum speedup is 1.58. Using five or more SPE threads for loop parallelization decreases efficiency. The reasons for the seemingly low speedup are the non-optimal coverage

1 worker, 1 bootsrtap, no LLP	28.71s
1 worker, 1 bootsrtap, 2 SPEs used for LLP	20.83s
1 worker, 1 bootsrtap, 3 SPEs used for LLP	19.37s
1 worker, 1 bootsrtap, 4 SPEs used for LLP	18.28s
1 worker, 1 bootsrtap, 5 SPEs used for LLP	18.10s
1 worker, 1 bootsrtap, 6 SPEs used for LLP	20.52s
1 worker, 1 bootsrtap, 7 SPEs used for LLP	18.27s
1 worker, 1 bootsrtap, 8 SPEs used for LLP	24.4s

Table 2. Execution time of RAxML when loop-level parallelism (LLP) is used in one bootstrap, across SPEs. The input file is 42_SC.

of loop-level parallelism (less than 90% of the original sequential code), the fine granularity of the loops, and the fact that many of the loops have global reductions, which constitute a bottleneck. Higher speedup from LLP in a single bootstrap can be obtained with larger input data sets. Alignments that have a larger number of nucleotides per organism have more loop iterations to distribute across SPEs [23].

Master SPE:

```
struct Pass pass[Num_SPE];

for(i=0; i < Num_SPE; i++){
  pass[i].sig[0] = 1;
     ...
  send_to_spe(i,&pass[i]);
}

for ( ... )
{
  /* see Figure 3 */
}

tr->likeli = sum;

for(i=0; i < Num_SPE; i++){
  while(pass[i].sig[1] == 0);
  pass[i].sig[1] = 0;
  tr->likeli += pass[i].res;
}

commit(tr->likeli);
```

Worker SPE:

```
struct Pass pass;

while(pass.sig[0]==0);
fetch_data();

for ( ... )
{
  /* see Figure 3 */
}

tr->likeli = sum;
pass.res = sum;
pass.sig[1] = 1;
send_to_master(&pass);
```

Figure 6. Parallelization of the loop from function `evaluate()` **of RAxML, shown in Figure 3. The left side shows the code executed by the master SPE, while the right side shows the code executed by a worker SPE.** `Num_SPE` **represents the number of SPE worker threads.**

5.4 Adaptive Scheduling of Task-Level and Loop-Level Parallelism

No single parallelization technique gives the best performance in all possible situations on Cell, a result which is expected given the variable degree of parallelism available in different components of parallel workloads and the heterogeneity of the Cell architecture.

We implemented a unified dynamic parallelization strategy, which exploits multiple layers of parallelism, by mixing and matching EDTLP with loop-level parallelization, under the control of the run-time system. We name this scheduling strategy multigrain parallelism scheduling (MGPS). The goal of MGPS is to exploit the best of two worlds (TLP and LLP), in response to workload characteristics. MGPS changes parallelization strategies and execution policies on the fly, while the program executes.

To illustrate the need for selectively combining TLP and LLP, we conduct a set of experiments, in which we generate a varying number of bootstraps in RAxML, ranging from 1 to 128, and use static EDTLP and hybrid EDTLP-LLP parallelization schemes. When LLP is used, each loop uses two or four SPEs, and the PPEs can execute four or two concurrent bootstraps respectively, using EDTLP. This leads to a static multigrain scheme (EDTLP-LLP), where LLP is activated when four or less MPI processes are active on the PPE. When LLP is deactivated, we use EDTLP to off-load to all 8 SPEs. The combination of LLP and EDTLP in the static multigrain model is not our final MGPS scheme, since it lacks dynamicity and assumes prior knowledge of runtime program properties. We are using it solely for illustrative purposes.

Figure 7 shows the results with a varying number bootstraps. The x-axis shows the number of performed bootstraps, and the y-axis shows the execution time in seconds. The EDTLP-LLP and EDTLP schemes are compared.

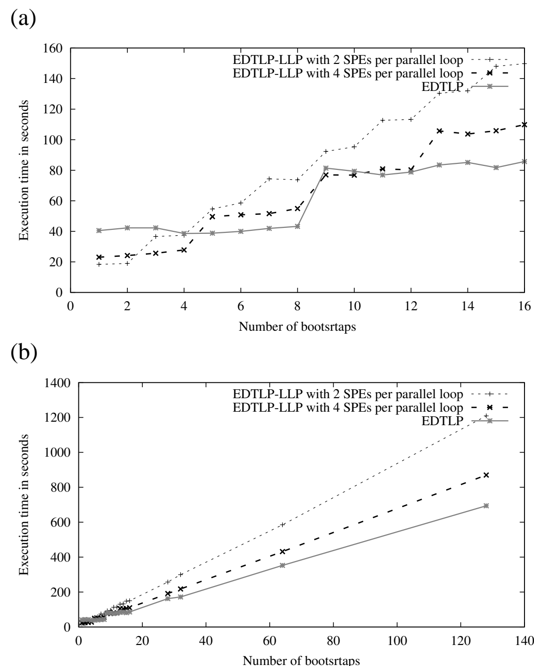

Figure 7. Comparison between the static EDTLP-LLP and EDTLP scheduling schemes. The input file is 42_SC. The number of ML trees created is (a) 1–16, (b) 1–128.

As expected, the hybrid model outperforms EDTLP when up to 4 bootstraps are executed, since only a combination of EDTLP and LLP can use more than 4 SPEs simultaneously (see Section 5.2). With 5 to 8 bootstraps, EDTLP activates 5 to 8 SPEs solely for task-level parallelism, leaving room for loop-level parallelism on at most 3 SPEs. This proves to be unnecessary, since the parallel execution time is determined by the length of the non-parallelized off-loaded tasks that remain on at least one SPE. In the range between 9 and 12 bootstraps, combining EDTLP and LLP selectively, so that the first 8 bootstraps execute with EDTLP and the last 4 bootstraps execute with the hybrid scheme is the best option. Note that this scheme is application-specific and requires an oracle to dictate the runtime system when to use EDTLP or EDTLP combined with LLP. Note also that the difference between EDTLP and the hybrid EDTLP-LLP scheme is smaller with 9 to 12 total bootstraps, than with 1 to 4 total bootstraps. In the former case LLP covers up to 11% (for 9 bootstraps), to 33% (for 12 bootstraps) of the parallel computation,

whereas in the latter case there are always enough SPEs so that the entire parallel computation benefits from LLP.

EDTLP becomes again the best choice with 13 to 16 bootstraps, by the same argument that justifies its superior performance with 5 to 8 bootstraps. As the number of bootstraps increases, the occasional benefit from LLP diminishes, since execution time is dominated by task-level parallelism.

Our experimental observations point to the direction of a dynamic and adaptive user-level scheduler to benefit from multigrain parallelism on Cell. We implemented such a scheduler, MGPS and tested its performance with RAxML. MGPS extends the EDTLP scheduler with an adaptive processor-saving policy. The scheduler is distributed, and it is attached to every MPI process in the application. The scheduler is invoked upon requests for task off-loading (arrivals) and upon completion of off-loaded tasks (departures). Initially, upon arrivals, the scheduler conservatively assigns one SPE to each off-loaded task, anticipating that the degree of TLP is sufficient to use all SPEs. Upon a departure, the scheduler checks the degree of task-level parallelism exposed by each MPI process (we will call it U), i.e. how many discrete tasks were off-loaded to SPEs while the departing task was executing, and how many SPEs were not used in the same period. This number reflects the history of SPE utilization from TLP and is used to switch between the EDTLP policy and the EDTLP-LLP policy. If $U \leq 4$, and T is the number of tasks waiting for off-loading, the scheduler activates LLP with $\lfloor \frac{8}{T} \rfloor$ SPEs assigned to the parallel loops of each task, if any. If $U > 4$, the scheduler retains the EDTLP policy, or deactivates LLP, if LLP was previously activated.

The scheduler is sensitive to the length of the history of TLP, maintained to calculate U. As a heuristic, we maintain a history of length equal to the number of SPEs. This gives the scheduler the opportunity of a hysteresis of up to 8 off-loaded tasks, before deciding whether to activate LLP. The MPI process that completes the 8th, 16th, ..., task evaluates U and signals all other processes to release the idle SPEs, i.e. all SPEs that were not used during the last window of 8 off-loads. Depending on the value of U, the scheduler triggers or deactivates LLP. The implementation of the scheduler is facilitated with a shared arena established between MPI processes, to exchange signals and keep track of busy and idle SPEs at any scheduling point (arrivals and departures).

The switching between EDTLP and LLP is enabled by keeping dual copies of each off-loaded function which includes at least one parallel loop. In the complete adaptive scheduling scheme, each off-loaded function has two or three copies, one PPE copy, one non parallelized SPE copy, and, if the function encapsulates parallel loops, a parallelized SPE copy. Having multiple executable copies of functions increases the total size of the PPE and the SPE code. However, multiple copies avoid the use of conditionals, which are particularly expensive on the SPEs.

A drawback of the scheduler is that it initially needs to monitor several off-loading requests from MPI processes, before making a decision for increasing or throttling LLP. If the off-loading requests from different processes are spaced apart, there may be extended idle periods on SPEs, before adaptation takes place. In practice, this problem appears rarely, first because applications spawn parallelism early in the code and this parallelism can be directly off-loaded to SPEs, and second because parallelism is typically spawned in bursts from all MPI processes. MGPS handles applications with static loop-level parallelism as well as applications with static hybrid parallelism, such as MPI/OpenMP applications. To schedule applications that do not off-load enough tasks to trigger adaptation, the scheduler uses timer interrupts.

We compare MGPS against the EDTLP scheduler and the static hybrid (EDTLP-LLP) scheduler, which uses an oracle for the future to guide decisions between EDTLP and EDTLP-LLP. Figure 8 shows the execution times of the MGPS, EDTLP-LLP and EDTLP schedulers with various RAxML workloads. The x-axis shows the number of bootstraps, while the y-axis shows execution time.

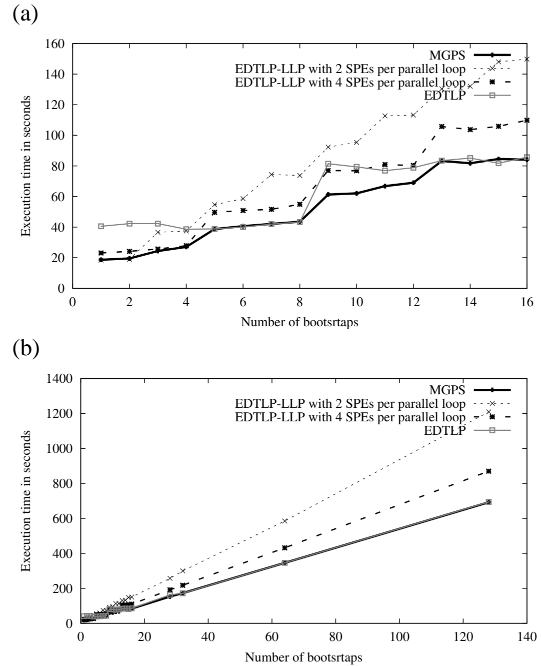

Figure 8. Comparison between the MGPS, EDTLP and static EDTLP-LLP schedulers. The input file is 42_SC. The number of ML trees created is (a) 1–16, (b)1–128. The curves of MGPS and EDTLP overlap completely in (b).

We observe benefits from using MGPS for up to 28 bootstraps, where LLP can be exploited by the scheduler in up to 4 concurrent bootstraps. Beyond 28 bootstraps, MGPS converges to EDTLP, and both are increasingly faster than the static multigrain EDTLP-LLP scheme, as the number of bootstraps increases. The loop-level parallel code in MGPS incurs additional overhead for loading the parallel loop code on idle SPEs, potentially replacing the image of an earlier off-loaded function on the SPEs and scheduling the loop. Code replacements happen whenever the runtime system needs to switch between a version with parallelized loops and a version of the same function without parallelized loops, or vice versa. This overhead is not noticeable in overall execution time. Somewhat to our surprise, this overhead is lower than the overhead of using conditionals to select between versions of each function loaded in the same SPE code image. This is an after-effect of the slow handling of branches on the SPEs.

5.5 Parallelizing Across Multiple Cells

Figure 9 shows the performance of the MGPS, EDTLP-LLP and the EDTLP schedulers with RAxML on two Cell processors that reside on a single blade. We use the same input file (42_SC) as in the single-processor experiments. The results are qualitatively identical to the results obtained with one Cell processor. The EDTLP-LLP model performs better with up to 8 bootstraps, since 8 additional SPEs are available across the two Cells for LLP. Beyond 8 bootstraps, task-level parallelism dominates and EDTLP performs better. MGPS outperforms both EDTLP-LLP and EDTLP.

The reader may point out that since RAxML needs 100 to 1,000 bootstraps for real-world biological analysis, multigrain paralleliza-

tion is obsolete. Our evaluation indicates that with more than 100 bootstraps, EDTLP is clearly the best option. The results of parallelization across two Cell processors provide a counter-argument. For a fixed number of bootstraps, two Cells deliver almost twice the performance of one Cell. As the application is scaled to multiple Cell Processors in the same blade or across blades, running fewer bootstraps per Cell is better than clustering bootstraps in as few Cells as possible. With 100 bootstraps, MGPS with multigrain (EDTLP-LLP) parallelism will outperform plain EDTLP if the bootstraps are distributed between four or more dual-Cell blades. Taking into account future system scaling, the MGPS scheme is justified in the range of interesting problem sizes for RAxML and at modest system scales.

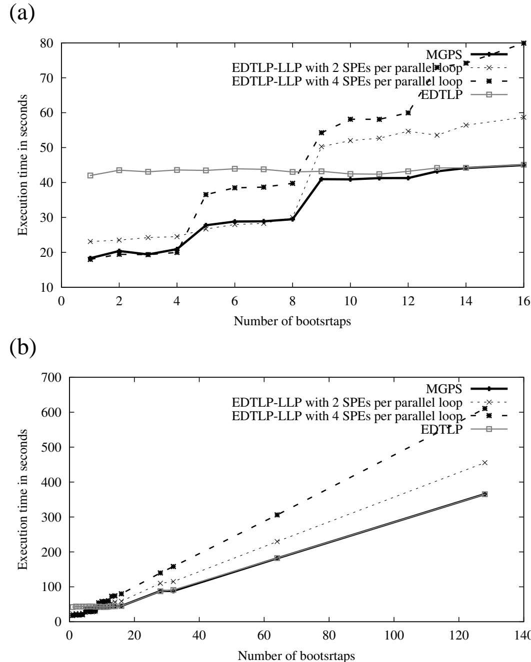

Figure 9. Comparison between MGPS, EDTLP and EDTLP-LLP on two Cell processors. The input file is 42_SC. The number of ML trees created is (a) 1–16, (b) 1–128. The curves of MGPS and EDTLP overlap completely in (b).

5.6 Comparison of Cell with Other Processors

As a last point in our evaluation, we compare the performance of the Cell implementation of RAxML and the MPI implementation of RAxML on two real microprocessors with multicore and SMT architecture:

- An Intel Pentium 4 Xeon with Hyper-threading technology (2-way SMT), running at 2GHz, with 8KB L1-D cache, 512KB L2 cache, and 1MB L3 cache.
- A 64-bit Power5. The Power5 is a quad-thread, dual-core processor with dual SMT cores running at 1.6 GHz, 32KB of L1-D and L1-I cache, 1.92 MB of L2 cache, and 36 MB of L3 cache.

For all experiments, we use 42_SC as an input file. Figure 10 illustrates execution time versus the number of bootsrtaps. While conducting the experiments on IBM Power5, we use both cores, and on each core we use both SMT execution contexts, i.e. a total of four MPI processes runs on the Power5 processor. Since one Intel Xeon processor has only two execution contexts, we use two Intel Xeon processors (lying on a 4-way SMP Dell PowerEdge 6650 server), and on each processor we use both execution contexts. This modification stirs the comparison in favor of the Xeon.

One Cell processor clearly outperforms the Intel Xeon by a large margin, even if two Xeons are used to run RAxML with the same problem size. Cell performs slightly (5–10%) better than the IBM Power5, once the problem size is scaled to 8 or more bootstraps. Although the margin of difference between Cell and Power5 is narrow, Cell has an edge over a general-purpose high-end processor such as Power5, since it also achieves better cost-performance and power-performance ratios.

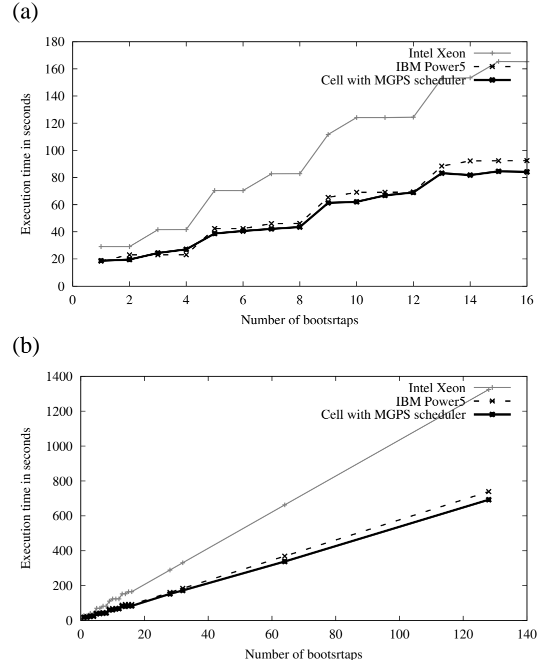

Figure 10. RAxML performance on different multithreaded and multicore microprocessors: Intel Xeon, IBM Power5 and Cell. The number of ML trees created is (a) 1–16, (b) 1–128.

6 Conclusions

We investigated issues, policies and mechanisms pertaining to scheduling multigrain parallelism on the Cell Broadband Engine. We proposed an event-driven task scheduling policy for Cell, striving for higher utilization via oversubscribing the PPE. We have explored the conditions under which loop-level parallelism within off-loaded code can be used. We have also proposed a comprehensive scheduling policy for combining task-level and loop-level parallelism autonomically within MPI code, in response to workload characteristics. Using a bio-informatics code with inherent multigrain parallelism as a case study, we have shown that our user-level scheduling policies outperform the native OS scheduler by a factor of 2.6, and they are able to transparently exploit the appropriate form and granularity of parallelism under widely varying execution conditions. Although our results use a single application case study, we believe they generalize to a broad range of applications, particularly those written in MPI or in the hybrid MPI/OpenMP model. Our scheduler is responsive to small and large degrees of task-level and data-level parallelism, at both fine and coarse levels of granularity.

In future work, we intend to incorporate memory-related criteria into our SPE scheduling policies. RAxML simplified the memory

management problem, since the major off-loaded functions have small memory footprints and leave enough space for data processing on the SPEs. At the same time, RAxML exhibits little sharing of data between tasks loaded on SPEs. We intend to eliminate the assumption of fixed-size SPE code footprints during exploration of scheduling policies in the future. We also plan to do more stress tests of our runtime system as more real-world application codes become available on Cell.

ACKNOWLEDGMENTS

This research is supported by the National Science Foundation (Grants CCR-0346867 and ACI-0312980), the U.S. Department of Energy (Grant DE-FG02-05ER2568), the Swiss Confederation Funding, the Barcelona Supercomputing Center, which granted us access to their Cell blades, and equipment funds from the College of Engineering at Virginia Tech.

7 REFERENCES

[1] PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual. http://www-306. ibm.com/chips/techlib.

[2] D.A. Bader, B.M.E. Moret, and L. Vawter. Industrial applications of high-performance computing for phylogeny reconstruction. In *Proc. of SPIE ITCom*, volume 4528, pages 159–168, 2001.

[3] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. *Technical Report, inTrace Realtime Ray Tracing GmbH, No inTrace-2006-001 (submitted for publication)*, 2006.

[4] Filip Blagojevic, Dimitrios S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Porting and Optimizing Phylogenetic Tree Construction on the Cell Broadband Engine. Technical report, Department of Computer Science, Virginia Tech, August 2006.

[5] M. Charalambous, P. Trancoso, and A. Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In *In Proceedings of the 10th Panhellenic Conference on Informatics (PCI 2005)*, pages 415–425, 2005.

[6] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation. *IBM developerWorks*, Nov 2005.

[7] Benny Chor and Tamir Tuller. Maximum likelihood of evolutionary trees: hardness and approximation. *Bioinformatics*, 21(1):97–106, 2005.

[8] T. Z. DeSantis, P. Hugenholtz, N. Larsen, M. Rojas, E. L. Brodie, K. Keller, T. Huber, D. Dalevi, P. Hu, and G. L. Andersen. Greengenes, a Chimera-Checked 16S rRNA Gene Database and Workbench Compatible with ARB. *Appl. Environ. Microbiol.*, 72(7):5069–5072, 2006.

[9] A. E. Eichenberger et al. Optimizing Compiler for a Cell processor. *Parallel Architectures and Compilation Techniques*, September 2005.

[10] B. Flachs et al. The Microarchitecture of the Streaming Processor for a CELL Processor. *Proceedings of the IEEE International Solid-State Circuits Symposium*, pages 184–185, February 2005.

[11] J. Felsenstein. Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolution*, 17:368–376, 1981.

[12] G. W. Grimm, S. S. Renner, A. Stamatakis, and V. Hemleben. A nuclear ribosomal dna phylogeny of acer inferred with maximum likelihood, splits graphs, and motif analyses of 606 sequences. *Evolutionary Bioinformatics Online*, 2006. to be published.

[13] S. Guindon and O. Gascuel. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Syst. Biol.*, 52(5):696–704, 2003.

[14] Nils Hjelte. Smoothed Particle Hydrodynamics on the Cell Brodband Engine. *Masters Thesis*, June 2006.

[15] Mike Kistler, Michael Perrone, and Fabrizio Petrini. Cell Multiprocessor Interconnection Network: Built for Speed. *IEEE Micro*, 26(3), May-June 2006. Available from http://hpc.pnl.gov/people/fabrizio/papers/ieeemicro-cell.pdf.

[16] R. E. Ley, J. K. Harris, J. Wilcox, J. R. Spear, S. R. Miller, B. M. Bebout, J. A. Maresca, D. A. Bryant, M. L. Sogin, and N. R. Pace. Unexpected diversity and complexity of the guerrero negro hypersaline microbial mat. *Appl. Envir. Microbiol.*, 72(5):3685 – 3695, May 2006.

[17] R.E. Ley, F. Backhed, P. Turnbaugh, C.A. Lozupone, R.D. Knight, and J.I. Gordon. Obesity alters gut microbial ecology. *Proceedings of the National Academy of Sciences of the United States of America*, 102(31):11070–11075, 2005.

[18] Bui Quang Minh, Le Sy Vinh, Arndt von Haeseler, and Heiko A. Schmidt. pIQPNNI: parallel reconstruction of large maximum likelihood phylogenies. *Bioinformatics*, 21(19):3794–3796, 2005.

[19] Barry Minor, Gordon Fossum, and Van To. Terrain renderin engine (tre), http://www. research.ibm.com/cell/whitepapers/tre.pdf. May 2005.

[20] Fabricio Petrini, Gordon Fossum, Mike Kistler, and Michael Perrone. Multicore Suprises: Lesson Learned from Optimizing Sweep3D on the Cell Broadbend Engine.

[21] C.E. Robertson, J.K. Harris, J.R.Spear, and N.R. Pace. Phylogenetic diversity and ecology of environmental Archaea. *Current Opinion in Microbiology*, 8:638–642, 2005.

[22] A. Stamatakis, T. Ludwig, and H. Meier. Raxml-iii: A fast program for maximum likelihood-based inference of large phylogenetic trees. *Bioinformatics*, 21(4):456–463, 2005.

[23] A. Stamatakis, M. Ott, and T. Ludwig. Raxml-omp: An efficient program for phylogenetic inference on smps. In *Proc. of PaCT05*, pages 288–302, 2005.

[24] Alexandros Stamatakis. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, page btl446, 2006.

[25] Alias Systems. Alias cloth technology demonstration for the cell processor, http://www.research.ibm.com/cell/ whitepapers/alias_cloth.pdf. 2005.

[26] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The Potentinal of the Cell Processor for Scientific Computing. *ACM International Conference on Computing Frontiers*, May 3-6 2006.

[27] Derrick Zwickl. *Genetic Algorithm Approaches for the Phylogenetic Analysis of Large Biologiical Sequence Datasets under the Maximum Likelihood Criterion.* PhD thesis, University of Texas at Austin, April 2006.