RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine

Filip Blagojevic[1], Alexandros Stamatakis[2],
Christos D. Antonopoulos[3] and Dimitrios S. Nikolopoulos[1]

[1]Center for High-end Computing Systems
Department of Computer Science
Virginia Tech

[2] School of Computer &
Communication Sciences
Swiss Federal Institute of Technology

[3] Department of Computer Science
College of William and Mary

Abstract

Phylogenetic tree reconstruction is one of the grand challenge problems in Bioinformatics. The search for a best-scoring tree with 50 organisms, under a reasonable optimality criterion, creates a topological search space which is as large as the number of atoms in the universe. Computational phylogeny is challenging even for the most powerful supercomputers. It is also an ideal candidate for benchmarking emerging multiprocessor architectures, because it exhibits various levels of fine and coarse-grain parallelism. In this paper, we present the porting, optimization, and evaluation of RAxML on the Cell Broadband Engine. RAxML is a provably efficient, hill climbing algorithm for computing phylogenetic trees based on the Maximum Likelihood (ML) method. The algorithm uses an embarrassingly parallel search method, which also exhibits data-level parallelism and control parallelism in the computation of the likelihood functions. We present the optimization of one of the currently fastest tree search algorithms, on a real Cell blade prototype. We also investigate problems and present solutions pertaining to the optimization of floating point code, control flow, communication, scheduling, and multi-level parallelization on the Cell.

1 Introduction

Phylogenetic (evolutionary) tree construction is one of the grand-challenge problems in computational biology. A phylogenetic tree depicts the evolutionary relationships between organisms, starting from a multiple alignment of DNA or AA sequences (taxa) representing organisms. The problem is intractable under the ML criterion [7].

Recent advances in high-performance computational biology enabled the construction of parallel, heuristic algorithms for the inference of phylogenetic trees. RAxML-VI-HPC is such a parallel algorithm, based on the Maximum Likelihood (ML) method. The original RAxML algorithm uses a rapid hill climbing search heuristic, which is able to infer large trees — in the order of 1,000 organisms— with low time and space requirements [28]. RAxML uses an embarrassingly parallel master-worker algorithm for non-parametric bootstrapping and multiple inference on distinct reasonable randomized starting trees (random stepwise addition sequence Maximum Parsimony trees [30]) in order to search for the best-known ML tree. RAxML's master-worker scheme is implemented using MPI. RAxML-VI-HPC has been further parallelized with OpenMP, to exploit the inherent loop-level parallelism of the likelihood functions. The shared-memory parallelization of the algorithm scales well and achieves good cache performance with inputs comprising large multi-gene alignments [31].

In this paper, we present the porting and optimization of RAxML on a real IBM Cell microprocessor. The Cell Broadband Engine [11], as

it became known in the popular press, has been developed jointly by Sony, Toshiba, and IBM. Although originally intended as a processor for Sony PlayStation3, Cell is a general-purpose architecture, offering a unique assembly of thread-level and data-level parallelization options. Cell provides eight execution cores, called Synergistic Processing Elements (SPEs), each equipped with a vector execution unit and an extended vector ISA. The chip also includes an SMT PowerPC processor, called the Power Processing Element (PPE), which runs Linux and operates either as a standalone general-purpose processor, or as a front-end controller for the SPEs. Cell has an aggressive memory and on-chip network architecture with a maximum bandwidth of over 200 Gigabytes/s. The PPE, SPEs and interconnect are all packaged on a single, thumb-size die, operating at the upper range of existing processor frequencies (3.2 GHz for current models, projected to run at more than 5 GHz in the near future [33]) and power consumption comparable to that of mobile processors [33].

Research on programming models, runtime environments, compiler support and application adaptation on Cell is highly relevant. The Cell has officially been adopted by IBM as the processor of choice for building a machine with sustained Petaflop performance before the end of 2008 [1].

This paper makes the following contributions: I. We present a detailed empirical optimization process of RAxML on Cell. We use a real Cell blade and Cell's native programming toolkits for this study. RAxML is seemingly an ideal target for Cell. The code is embarrassingly parallel, and each parallel task includes loop-level and SIMD-level parallelism, which can be exploited via multithreading, vectorization or a combination of these two techniques. Optimizing RAxML on Cell is a non-trivial exercise, and one which is unlikely to be done automatically, without some level of support from the programmer and the runtime environment.

II. We quantify Cell-specific code optimizations and assess their impact, using RAxML. We find that merely exposing the multi-level parallelism of RAxML to Cell is insufficient for high performance. Both conventional and unconventional optimizations are important to accelerate program execution. Conventional Cell-specific optimizations include the use of optimized numerical libraries for the SPEs, double-buffering for communication/computation overlap, vectorization of floating point code, multi-level parallelization and offloading as much computation as possible from the PPE to the SPEs. Less common optimizations include the vectorization of conditional statements, asynchronous communication through direct SPE memory accesses, and interleaved event-driven scheduling of tasks across SPEs. Somewhat surprisingly, the less common optimizations yield higher performance improvements than the common optimizations. We find that event-driven task scheduling and a customized casting and vectorization scheme for complex conditionals are the most effective optimizations for RAxML on Cell.

III. We find that multi-level parallelization on Cell is both feasible and necessary, however its exploitation is intricate. Depending on the input, RAxML can exploit two or three layers of parallelism, with two layers of parallelism (task-level parallelism across SPEs and task vectorization within each SPE) being more beneficial for large and realistic workloads and three layers of parallelism (task-level parallelism across SPEs, loop-level parallelization of tasks across SPEs and vectorization of blocks of loop iterations within SPEs) being beneficial for workloads with a low degree (less than or equal to four) of task-level parallelism.

IV. We present a comparison between Cell, a cutting-edge multicore microprocessor (IBM Power5) and a mature multithreaded microprocessor (Intel Xeon with HT technology), using RAxML. To the best of our knowledge this is one of the first such studies, using a real-silicon

[1] See http://www.hpcwire.com/hpc/893353.html and numerous other popular press articles published in September 2006.

Cell prototype. The results demonstrate the superiority of Cell as a platform for high-end computing and grand-challenge applications.

The rest of this paper is organized as follows: Section 2 summarizes related work on programming support for the Cell processor and studies of computational biology codes on emerging parallel architectures. Section 3 presents RAxML-VI-HPC, the parallel version of RAxML for distributed and shared memory systems. Section 4 outlines the Cell architecture. Section 5 presents our step by step RAxML porting and optimization process, along with experimental results for each step of the process. Section 6 compares Cell with the IBM Power5 and Intel Xeon processors. Section 7 concludes the paper.

2 Related Work

Since the introduction of Cell for general-purpose computing, several researchers engaged in analyzing the performance of the processor and developing compiler and programming support. Kistler et. al [17] provide a performance analysis of the Cell's on-chip interconnection network, including DMA latencies and bandwidth. Williams et. al [34] presented an analytical framework to predict performance of code written for Cell. They exercised their model using small linear algebra kernels and, driven by their observations, they proposed microarchitectural extensions to improve double-precision floating point performance on the Cell.

Eichenberger et. al [9] presented several compiler techniques targeting automatic generation of highly optimized Cell code. The techniques include compiler-assisted memory alignment, branch prediction, SIMD parallelization, and OpenMP task level parallelization. This work presented also a compiler-controlled software cache. Our work departs in that it considers optimizations which may be hard to derive automatically in a compiler, such as casting and vectorization of conditionals, dynamic multi-level parallelization, event-driven task scheduling, and communication optimizations.

Phylogenetic tree construction has attracted considerable attention from the high-performance computing community, due to the computational challenges of the problem. RAxML has already been studied on distributed memory architectures [29], shared-memory multiprocessors [27] and graphics processing units [5]. Other researchers have studied phylogenetic tree construction on shared-memory parallel architectures using parsimony-based approaches [2], and distributed memory multiprocessors using maximum likelihood methods [32].

3 RAxML-VI-HPC

RAxML-VI-HPC (v2.2.0) (Randomized Axelerated Maximum Likelihood version VI for High Performance Computing) [29] is a program for large-scale ML-based (Maximum Likelihood [12]) 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 [3] for a summary). An example for the evolutionary tree of the monkeys and the homo sapiens is provided in Figure 1.

The fundamental algorithmic problem computational phylogeny faces consists in the immense amount of alternative tree topologies

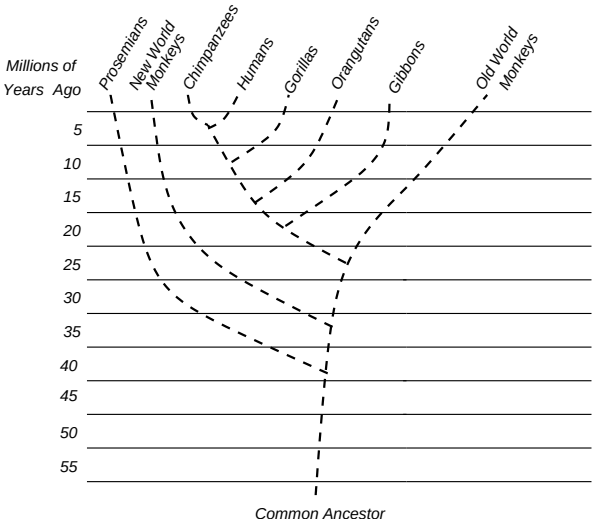

FIGURE 1: Phylogenetic tree representing the evolutionary relationship between monkeys and the homo sapiens.

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 computing 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 [21], PHYML [14], GARLI [35] and RAxML [26, 29].

Some of the largest published ML-based biological analyses to date have been conducted with RAxML [13, 18, 19, 22]. 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 [29] 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 [27]. 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 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 [13] 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 [26].

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 BE

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 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. 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 between 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 amounts 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 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 Porting and Optimizing RAxML on Cell

We adapted RAxML to Cell in three steps: we ported the MPI code on the PPE; we offloaded

the most time-consuming parts of each MPI process on the SPEs; we optimized the SPE code using vectorization of computation, a specialized casting transformation coupled with vectorization of control statements, and communication optimizations; lastly, we developed multi-level parallelization schemes across and within the SPEs in selected cases, as well as a scheduler for effective simultaneous task, loop, and vector-level parallelization of the application. We outline these optimizations in the following sections.

All results reported in this paper are obtained from a real dual-Cell Blade, located at the Barcelona Supercomputing Center. Each Cell processor has a PPE element and eight SPE elements. Each PPE element is a 2-way SMT Power architecture processor, running at 3.2 GHz with 512 MB XDR RAM. The size of the first level instruction and data caches is 32 KB, while the size of the second level cache is 512 KB. The operating system is Fedora Core 5 and Linux kernel version 2.6.16 with Cell-specific kernel patches. To compile our code, we used Toolchain 4.0.2.

5.1 Porting MPI Code

In the initial port, we assigned one MPI process to each thread on the PPE. Since the PPE is a dual-threaded processor, up to two processes can concurrently offload tasks from the PPE to SPEs. We will see later that this naïve porting strategy underutilizes the SPEs, since two MPI processes do not expose enough task-level parallelism for all 8 SPEs of Cell. For this reason, we introduced both loop-level parallelization of tasks across SPEs and a scheduler which multiplexes more than two MPI processes on the PPE using an event-driven model, in order to expose more task-level parallelism from the application.

5.2 Function Off-loading

We profiled the application using gprofile to identify the computationally intensive functions that could be candidates for offloading and optimization on SPEs. We used an IBM Power5 processor for profiling RAxML. For both, profiling and all benchmarking runs of RAxML presented in this paper, we used the input file 42_SC, which contains 42 organisms, each represented by a DNA sequence of 1167 nucleotides. The number of distinct data patterns in a DNA alignment is on the order of 250.

On the IBM Power5, executing the sequential version of RAxML with the 42_SC input file, we find that 98.77% of the total execution time is spent in three functions: 76.8% in newview() - which computes the partial likelihood vector [12] at an inner node of the phylogenetic tree, 19.16% in makenewz() - which optimizes the length of a given branch with respect to the tree likelihood using the Newton–Raphson method, and 2.37% in evaluate() - which calculates the Log Likelihood score of the tree at a given branch by summing over the partial likelihood vector entries. Note that the log likelihood value is the same at all branches of the tree if the model of nucleotide substitution is time-reversible [12, 24]. These functions are the best candidates for offloading on SPEs.

The prerequisite for computing evaluate() and makenewz() is that the likelihood vectors at the nodes to the right and left of the branch have been computed. Thus, makenewz() and evaluate() initially make calls to newview() before they can execute their own computation. The newview() function at an inner node p calls itself recursively when the two children r and q are not tips (leaves) and the likelihood array for r and q has not already been computed. Consequently, the first candidate for offloading is the newview() function. Although makenewz() and evaluate() are both taking a smaller portion of the execution time than newview(), offloading these two functions can also bring significant speedup (see Section 5.2.7). Besides the fact that each function can be executed faster

on an SPE, having all three functions offloaded to an SPE significantly reduces the amount of PPE-SPE communication.

In order to have a function executed on an SPE, we spawn an SPE thread at the beginning of each MPI process. The thread executes the offloaded function upon receiving a signal from the PPE and returns the result back to the PPE upon completion. To avoid excessive overhead from repeated spawning and joining of threads, threads remain bound on SPEs and perform a busy-wait for the PPE signal to start executing a function. Furthermore, we load the code of all three offloaded functions on each SPE, such that each thread can execute any of the functions on demand, including nested combinations of these functions (makenewz() with calls to newview(), and evaluate()). This decision imposes a trade-off, since the limited capacity of the local storage on the SPEs and the fact that local storage is used as a unified instruction and data cache, prevent arbitrary function offloading. This proves not to be a problem in RAxML, where the code footprints of the offloaded functions are small enough (117 Kbytes in total) to fit in the local storage and still leave 139 Kbytes free for stack, heap and static data.

5.2.1 Optimizing Off-Loaded Functions

The discussion in this Section refers to function newview(), which is the most computationally expensive part of the code. Table 1 summarizes the execution times of RAxML before and after newview() is offloaded. The first column shows the number of workers (MPI processes) used in the experiment and the amount of work performed.

As shown in Table 1, merely offloading a function causes performance degradation. We used the SPE decrementer register to measure the time spent in the SPE thread by newview(). We profiled the new code in order to get a better understanding of the major bottlenecks. Inside newview(), we identified 4 parts where the function spent almost its entire time: The first includes math library functions such as exp() and

(a)

1 worker, 1 bootstrap	36.9s
2 workers, 8 bootstraps	207.67s
2 workers, 16 bootstraps	427.95s
2 workers, 32 bootstraps	824s

(b)

1 worker, 1 bootstrap	106.37s
2 workers, 8 bootstraps	459.16s
2 workers, 16 bootstraps	915.75s
2 workers, 32 bootstraps	1836.6s

TABLE 1: Execution time of RAxML (in seconds). The input file is 42_SC: (a) The whole application is executed on the PPE, (b) newview() is offloaded on one SPE.

log(). The exp() function is required to compute the transition probabilities of the nucleotide substitution matrix for the branches from the root of a subtree to its descendants. The log() function is used to scale the branch lengths for numerical reasons [12, 23]; the second part includes a large if(...) statement with a conjunction of four arithmetic comparisons, that is used to check if small likelihood vector entries need to be scaled to avoid numerical underflow (similar operations are used in every ML implementation); the third includes DMA transfers; the fourth includes the large loops that perform the actual likelihood vector calculation. In the next few sections we describe the techniques used to optimize newview(). Similar techniques were applied to the other offloaded functions.

5.2.2 Math functions

The average number of floating point operations during a single invocation of newview() is 25,554, for the 42_SC input. 65% of these operations are multiplications and 34% are additions. The exp() function is called approximately 150 times. Although it represents a very small portion of the total number of floating point operations, the exp() function takes 50% of the total SPE time. We removed the math library function exp() from the code and used the exponential function provided by the exp.h

header file which comes with the Cell SDK 1.1. The new exp() function implements a numerical method for the exponent calculation. The execution time after replacing exp() is shown in Table 2. Following replacement of exp() with an optimized implementation, execution time is reduced by 37%–41%, for the 42_SC input.

1 worker, 1 bootstrap	62.8s
2 workers, 8 bootstraps	285.25s
2 workers, 16 bootstraps	572.92s
2 workers, 32 bootstraps	1138.5s

TABLE 2: Execution time of RAxML when we use the exp() function from the SDK library. The input file is 42_SC.

5.2.3 Converting if() statements

Function newview() is always invoked at an inner node of the tree (p) which is at the root of a subtree. The main computational kernel of newview() has a switch statement which selects one out of four paths of execution. If one or both descendants r and q of p are tips (leaves) the computations of the main for-loop in newview() can be simplified which leads to significant performance improvements [29]. Thus, there are distinct implementations of the main computational part of newview() for the case that r and q are tips, r is a tip, q is a tip, or r and q are both inner nodes.

Each of the above paths leads to a distinct—highly optimized—version of the loop which performs the actual likelihood vector calculations. Each iteration of this loop executes the previously mentioned large if() statement (Section 5.2.1), to check for likelihood scaling. Mispredicted branches in the compiled code for this statement incur a penalty of approximately 20 cycles [15]. We profiled newview() and found that 45% of the function execution time is spent in this conditional statement. Furthermore, almost all the time is spent in checking the condition, while negligible time is spent in the body of the conditional statement. The problematic conditional statement is shown below, where ml is a constant, and all operands are double precision floating point numbers.

```
if (ABS(x3->a) < ml && ABS(x3->g) < ml
&& ABS(x3->c) < ml && ABS(x3->t) < ml) {. . .}
```

This statement is a challenge for a branch predictor, since it implies 8 conditions, one for each of the four ABS() macros and four comparisons against the minimum likelihood value constant (ml).

On an SPE, comparing integers can be significantly faster than comparing doubles, since integer values can be compared using the SPE intrinsics. Although the current SPE intrinsics support only comparison of 32-bit integer values, the comparison of 64-bit integers is also possible by combining different intrinsics that operate on the 32-bit integers. The current spu-gcc compiler automatically optimizes an integer branch using the SPE intrinsics. To optimize the problematic branches we exploited the fact that integer comparison is faster than floating point comparison on an SPE.

According to the IEEE standard, numbers represented in float and double formats are "lexicographically ordered", i.e., if two floating point numbers in the same format are ordered, then they are ordered the same way when their bits are reinterpreted as Sign-Magnitude integers [16]. In other words, instead of comparing two floating point numbers we can interpret their bit pattern as integers, and do an integer comparison. The final outcome of comparing the integer interpretation of two doubles (floats) will be the same as comparing their floating point values, as long as one of the numbers is positive. In our case, all operands are positive, consequently instead of floating point comparison we can perform an integer comparison.

To get an absolute value of a floating point number, we used the spu_and() logic intrinsic, which performs vector bit-wise AND operation. We set the left most bit of a floating point number to be one. If the number is already positive, nothing will change, since the most significant bit is already one. In this way, we avoid us-

ing ABS(), which uses a conditional statement to check if the operand is greater than or less than 0. After getting absolute values of all the operands involved in the problematic if() statement, we cast each operand to an unsigned long long value and perform the comparison.

After optimizing the conditional statements, their portion of the execution time in the function newview() is only 6%, as opposed to 45% with the original conditional statement. The total execution time also improves significantly as shown in Table 3. Casting and vectorization of the if() statements reduced the execution time of RAXML with the 42_SC input by a further 19%–21%, after optimization of mathematical functions with SDK.

1 worker, 1 bootstrap	49.3s
2 workers, 8 bootstraps	230s
2 workers, 16 bootstraps	460.43s
2 workers, 32 bootstraps	917.09s

TABLE 3: Execution time of RAxML after the floating-point conditional statement is transformed to an integer conditional statement and vectorized. The input file is 42_SC.

5.2.4 Double Buffering and Memory Management

Depending on the size of the input alignment, the major calculation loop (the loop that performs the calculation of the likelihood vector) in newview() can execute up to 50,000 iterations. The number of iterations is directly related to the alignment length. The loop operates on large arrays, and each member in the arrays is an instance of a likelihood_vector structure. The arrays are allocated dynamically at runtime. Since there is no limit on the size of these arrays, we are unable to keep all the members of the arrays in the local storage of SPEs. Instead, we strip-mine the arrays, by fetching a few array elements to local storage at a time, and execute the corresponding loop iterations on each batch of elements. We use a 2 KByte buffer for caching likelihood vectors, which is enough to store the data needed for 16 loop iterations. It should be noted that the space used for buffers is much smaller than the size of the local storage. Considering that the loop actually executes a recursion, we opted to keep the buffer small, so that the recursion can be executed without overflowing the local storage. This is another example of the trade-off between code loading and data caching on the Cell SPEs. Recursive function calls in general, necessitate the use of manually managed code overlays on the Cell. We have not experimented with this option, relying instead on careful control of the code footprint of the offloaded functions to avoid overlays, even in the presence of recursion.

In the original code where SPEs wait for all DMA transfers, the idle time accounts for 11.4% of the total execution time of newview(). We eliminated this waiting time by using double buffering to overlap DMA transfers with computation. The total execution time of the application after applying double buffering and tuning the transfer size (set to 2 KBytes) is shown in Table 4.

1 worker, 1 bootstrap	47s
2 workers, 8 bootstraps	220.92s
2 workers, 16 bootstraps	441.39s
2 workers, 32 bootstraps	884.47s

TABLE 4: Execution time of RAxML with double buffering applied to overlap DMA transfers and computation. The input file is 42_SC.

Double buffering and communication-computation overlap improve execution time by 4%–5%, on top of the improvements obtained with optimization of mathematical functions and casting and vectorization of conditional statements.

5.2.5 Vectorization

All calculations in newview() are executed in two different loops. The first loop has a small

trip count (typically 4–25 iterations) and computes the individual transition probability matrices (see Section 5.2.1) for each distinct rate category of the CAT or Γ models of rate heterogeneity [25]. Each iteration executes 36 double precision floating point operations. The second loop computes the likelihood vector. Typically, the second loop has a large trip count, which depends on the number of distinct data patterns in the data alignment. For the 42_SC input file, the loop has 228 iterations, and executes 44 double precision floating point operations per iteration. Each SPE on the Cell is capable of exploiting data parallelism via vectorization. The SPE has vector registers, each of which can store 128 bits. This is enough space for two double precision floating point elements. We vectorized the two dominant loops in newview(). Here, we present a simplified explanation of our vectorization strategy. As an example we are using the larger loop. The smaller loop is optimized in a similar way.

Figure 2 illustrates the larger loop (showing the instructions that dominate the body of the loop). The variables x1->a, x1->c, x1->g, x1->t, belong to the same C structure (likelihood_vector) and occupy contiguous memory locations. Only three of these variables are multiplied by the elements of the array left. This makes vectorizing more difficult, since the code requires vector construction instructions such as spu_splats(). Obviously, there are many different possibilities for vectorizing this code. The scheme shown in Figure 2 is the one that achieved the best performance in our experiments. Note that due to involved pointer arithmetic on dynamically allocated data structures, automatic vectorization of this code may be challenging for a compiler. After vectorization, the number of floating point instructions in the body of the loops dropped from 36 to 24 for the first loop, and from 44 to 22 for the second loop. Vectorization added a total of 25 new instructions for creating vectors.

Without vectorization, newview() spends 19.57 seconds (or 69.4% of its total execution time) in the two loops. After vectorization, the time spent in loops drops to 11.48 seconds, and accounts for 57% of the total execution time of newview(). The total execution time of the application is reduced by 9%–13% due to vectorization, with all optimizations discussed so far already integrated in the code. Table 5 shows execution times after vectorization. Somewhat surprisingly, vectorization of floating point code has a lesser impact on performance than vectorization of control statements in RAxML.

1 worker, 1 bootstrap	40.9s
2 workers, 8 bootstraps	195.7s
2 workers, 16 bootstraps	393s
2 workers, 32 bootstraps	800.9s

TABLE 5: Execution time of RAxML after vectorization. The input file is 42_SC.

5.2.6 PPE-SPE communication

Although most of the total execution time of RAxML is spent in newview(), the granularity of this function is actually fine. For the 42_SC input, the newview() function is invoked 230,500 times and the average execution time per invocation is 71μs. In order to invoke an offloaded function, the PPE needs to send a signal to an SPE. Also, after an offloaded function completes its execution, it needs to send the result back to the PPE.

In our first implementation, the communication between the PPE and SPEs was implemented through mailboxes. We found that PPE-SPE communication can be significantly improved if it is performed through main memory and SPE local storage instead of mailboxes. Every time the PPE needs to signal an SPE, instead of using mailboxes, the PPE changes a variable that resides in the local storage of the specific SPE. In the same way, every time an SPE sends data to the PPE, instead of using mailboxes, the SPE commits data directly to main memory. This optimization improves execution

```
for( . . . )
{
  ump_x1_0 = x1->a;
  ump_x1_0 += x1->c * *left++;
  ump_x1_0 += x1->g * *left++;
  ump_x1_0 += x1->t * *left++;

  ump_x1_1 =  x1->a;
  ump_x1_1 += x1->c * *left++;
  ump_x1_1 += x1->g * *left++;
  ump_x1_1 += x1->t * *left++;
  . . .
}
```

```
for( . . . )
{
  a_v = spu_splats(x1->a);
  c_v = spu_splats(x1->c);
  g_v = spu_splats(x1->g);
  t_v = spu_splats(x1->t);
  l1 = (vector double)(left[0],left[3]);
  l2 = (vector double)(left[1],left[4]);
  l3 = (vector double)(left[2],left[5]);
  ump_v1[0] = spu_madd(c_v,l1,a_v);
  ump_v1[0] = spu_madd(g_v,l2,ump_v1[0]);
  ump_v1[0] = spu_madd(t_v,l3,ump_v1[0]);
  . . .
}
```

FIGURE 2: Example of one of the large loops in newview(): Non–vectorized code shown on the left, vectorized code shown on the right. spu_madd() multiplies the first two arguments and adds the result to the third argument. spu_splats() creates a vector by replicating a scalar element.

time by 2%–11%. Table 6 shows the new execution times, including all optimizations discussed so far and direct memory to memory communication, for the 42_SC input. It is interesting to note that direct memory-to-memory communication is an optimization which scales with parallelism on Cell, i.e. its performance impact grows as the code uses more SPEs. As the number of workers and bootstraps executed on the SPEs increases, the code becomes more communication-intensive, due to the fine granularity of the offloaded functions. Fast communication therefore becomes critical.

1 worker, 1 bootstrap	39.9s
2 workers, 8 bootstraps	180.46s
2 workers, 16 bootstraps	357.08s
2 workers, 32 bootstraps	712.2s

TABLE 6: Execution time of RAxML after optimizing communication to use direct memory-to-memory transfers. The input file is 42_SC.

5.2.7 Offloading More Functions

After offloading and optimizing newview(), we proceeded with offloading the next two most expensive functions: makenewz() and evaluate(). As mentioned earlier, we offloaded the functions in a single code module loaded on the SPEs. The advantage of having a single module is that it can be loaded to the local storage once, when an SPE thread is created. The offloaded code remains in local storage during the entire execution of the program, and the cost of loading code on the SPEs is amortized over the entire execution. By having all the three functions offloaded to an SPE we reduce the communication between the PPE and SPEs. When newview() is called by makenewz() or evaluate(), there is no need for any PPE-SPE communication (all functions are executed on the same SPE).

After offloading and optimizing the three most time-consuming functions, performance improves by 31%–38%, compared to the optimized code with only one of the functions (newview()) offloaded. A more important implication is that after offloading and optimizing all three functions, the sequential RAxML code with the major part of it offloaded on one SPE is faster than the sequential code executed exclusively on the PPE by 25%. Function offloading is also an optimization which scales with parallelism. When more than one MPI processes are used and more than one bootstraps are of-

floaded to SPEs by each process, the gains from offloading reach 47%. Table 7 illustrates execution times after complete function offloading.

1 worker, 1 bootstrap	27.7s
2 workers, 8 bootstraps	112.41s
2 workers, 16 bootstraps	224.69s
2 workers, 32 bootstraps	444.87s

TABLE 7: Execution time of RAxML after offloading and optimizing three functions: newview(), makenewz() and evaluate(). The input file is 42_SC.

5.3 Multilevel Loop-Level and Task-Level Parallelization

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 eight SPEs, via concurrent function offloading. We considered two programming models in order to exploit all eight SPEs on Cell. The first is loop-level parallelization (LLP) within the offloaded functions and loop distribution across SPEs. This model is similar to the OpenMP programming model used on conventional shared memory multiprocessors. It exposes a total of three levels of parallelism, with tasks distributed between some (up to four) SPEs, loops in each task distributed across the remaining SPEs, and blocks of loop iterations in each task vectorized within SPEs. The second model is event-driven task-level parallelization (EDTLP), in which the PPE scheduler oversubscribes the PPE with more than two MPI processes, to increase the availability of tasks for SPEs. More specifically, the scheduler multiplexes more than two MPI processes on the two threads of the PPE and enforces a context-switch whenever an MPI process offloads a function on the SPE. The EDTLP model exposes two levels of parallelism, with tasks distributed between all SPEs, and each task vectorized within an SPE. We implemented both parallelization models and observed that there is no single model that performs the best in all cases [4]. Consequently, we also implemented a dynamic parallelization scheme, named MGPS (Multi-grain Parallelism Scheduling) [4], where the LLP and the EDTLP models are combined.

In the MGPS model, the scheduler decides on-the-fly which parallelization model (EDTLP, LLP, or both) the application should use at any instant during execution. The decision is made at runtime, and it is based on the amount of work that the application performs. If there is enough task-level parallelism to keep eight SPEs busy via concurrent function offloading, the scheduler multiplexes up to eight MPI processes on the PPE, using the "switch-on-offload,, scheduling policy. More MPI processes are served in batches of eight. If there is not enough work to keep the eight SPEs busy, (i.e. when the number of bootstraps in RAxML is less than eight), the idle MPI processes are suspended, and the remaining active MPI processes use the idle SPEs for loop-level parallelization of the offloaded functions. Loop-level parallelism can be extracted from up to four simultaneously executing MPI processes, using two SPEs per loop.

Using the MGPS model we were able to further reduce the execution time of RAxML, as shown in Table 8. The number of workers used by MGPS is determined at runtime. At startup, MGPS uses eight workers scheduled with EDTLP, anticipating that there is enough task-level parallelism to utilize the SPEs. When the number of remaining bootstraps decreases below eight, the scheduler suspends idle workers and signals the rest of the workers to use loop-level parallelism. The mechanisms, policies and ramifications of the EDTLP and MGPS schedulers, as well as an application-independent implementation of these schedulers on Cell are discussed elsewhere [4]. In this paper we only report on the net impact of these schedulers on the performance of RAxML. The schedulers have a profound impact on performance. They reduce execution time by 36% in the one-bootstrap case, due to loop-level paral-

lelization of one task across eight SPEs, and up to 63% with more bootstraps, due to simultaneous exploitation and orchestration of task-level and loop-level parallelism.

1 bootstrap	17.6s
8 bootstraps	42.18s
16 bootstraps	84.21s
32 bootstraps	167.57s

TABLE 8: Execution time of RAxML when the dynamic parallelization model (MGPS) is used. The input file is 42_SC. The number of workers is variable and is selected at runtime by the scheduler.

6 Performance Comparison with Other Platforms

As a last point in our evaluation, we compare the performance of the Cell implementation of RAxML to the MPI implementation of RAxML on two real multiprocessors based on multicore and SMT architectures:

- A 32-bit 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 IBM Power5 processor. The Power5 is a quad-thread, dual-core processor with dual SMT cores running at 1.65 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 3 illustrates execution time versus the number of bootstraps. While conducting the experiments on the 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 simultaneously on the Power5. 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 favors the Xeon platform.

One Cell processor clearly outperforms the Intel Xeon by a large margin (more than a factor of two), even if two Xeons are used to run RAxML with the same problem size. Cell performs 9%-10% better than the IBM Power5. Although the margin of difference between Cell and Power5 seems small, Cell has an edge over a general-purpose high-end processor such as Power5, since Cell appears to be significantly more power-efficient claiming nominal power consumption in the range of 27W to 43W for a 3.2 GHz model (used in this study) [33], as opposed to a reported 150W for the Power5 [20]. Note also that the computation uses double-precision floating point arithmetic, which is not optimized for Cell SPE pipelines. The use of single-precision arithmetic would widen the margin between Cell and the Power5.

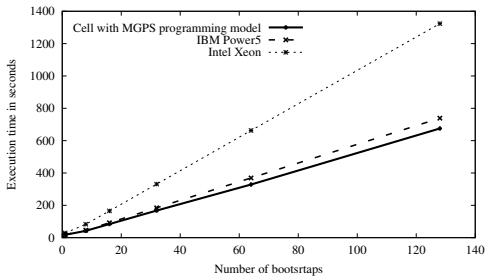

FIGURE 3: RAxML performance on different multi-threaded and multicore microprocessors: Intel Xeon, IBM Power5 and Cell. The number of ML trees created is 1, 8, 16, 32, 64 and 128.

7 Conclusions

In this paper, we presented the parallelization and optimization process of RAxML (Randomized Accelerated Maximum Likelihood), an important application from the domain of computational biology, on the Cell Broadband Engine. We explored a total of seven Cell-specific optimizations and the performance implications

of these optimizations: I) We offloaded the bulk of the maximum likelihood tree calculation on the SPEs; II) We replaced expensive mathematical functions with Cell-specific numerical implementations of the same functions; III) We casted and vectorized expensive conditional statements involving multiple, hard to predict conditions; IV) we used double buffering to overlap completely DMA transfers with computation; V) we vectorized the core of the floating point computation; VI) we optimized PPE-SPE communication using direct memory-to-memory transfers; VII) we exploited task and loop-level parallelism dynamically, by oversubscribing the PPE and exploiting loop-level parallelism when the task-level parallelism exposed by the program leaves SPEs unused. In total, starting from an optimized version of RAxML for conventional uniprocessors and multiprocessors, we were able to boost performance on Cell by more than a factor of five and bring it to a higher level than the best performance achieved by the leading current multicore processors.

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.

References

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

[2] D. Bader, V. Chandu, and M. Yan. ExactMP: An Efficient Parallel Exact Solver for Phylogenetic Tree Construction using Maximum Parsimony. In *Proc. of the 2006 International Conference on Parallel Processing*, pages 65–72, Columbus, OH, August 2006.

[3] 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.

[4] Filip Blagojevic, Dimitrios S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Dynamic Multigrain Parallelization on the Cell Broadband Engine. Technical report, Department of Computer Science, Virginia Tech, September 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] D. Pham et al. The Design and Implementation of a First Generation Cell Processor. *Proc. Int'l Solid-State Circuits Conf. Tech. Digest, IEEE Press*, pages 184–185, 2005.

[12] J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol.*, 17:368–376, 1981.

[13] 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.

[14] 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.

[15] IBM. Cbe_tutorial_v1.1. 2006.

[16] W. Kahan. Lecture notes on the status of ieee standard 754 for binary floating-point arithmetic. 1997.

[17] 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.

[18] 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.

[19] 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.

[20] Sun Microsystems. Sun UltraSPARC T1 Cool Threads Technology. December 2005. http://www.sun.com/aboutsun/media/presskits/networkcomputing05q4/T1Infographic.pdf.

[21] 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.

[22] 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.

[23] A. Stamatakis. *Distributed and Parallel Algorithms and Systems for Inference of Huge Phylogenetic Trees based on the Maximum Likelihood Method*. PhD thesis, Technische Universitt Mnchen, Germany, October 2004.

[24] A. Stamatakis. Parallel and distributed computation of large phylogenetic trees. In Albert E.Zomaya, editor, *Parallel Computing for Bioinformatics and Computational Biology*, pages 327–346. John Wiley & Sons, 2006.

[25] A. Stamatakis. Phylogenetic models of rate heterogeneity: A high performance computing perspective. In *Proceedings of 20th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS2006)*, High Performance Computational Biology Workshop, Proceedings on CD, Rhodos, Greece, April 2006.

[26] 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.

[27] 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.

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

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

[30] Alexandros Stamatakis, Thomas Ludwig, and Harald Meier. RAxML-III: A Fast Program for Maximum Likelihood-based Inference of Large Phylogenetic Trees. *Bioinformatics*, 21(4):456–463, 2005.

[31] Alexandros Stamatakis, Michael Ott, and Thomas Ludwig. RAxML-OMP: An Efficient Program for Phylogenetic Inference on SMPs. *PaCT*, pages 288–302, 2005.

[32] C. Stewart, D. Hart, D. Berry, G. Olsen, E. Wernert, and W. Fischer. Parallel Implementation and Performance of FastDNAml – A Program for Maximum Likelihood Phylogenetic Inference. In *Proc. of Supercomputing'2001: High Performance Networking and Computing Conference*, Denver, CO, November 2001.

[33] D. Wang. Cell Microprocessor III. *Real World Technologies*, July 2005.

[34] 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.

[35] 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.