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
660 McBryde Hall, Blacksburg VA 24061
{filip,dsn}@cs.vt.edu

[2] School of Computer & Communication Sciences
École Polytechnique Fédérale de Lausanne
Station 14, Ch-1015 Lausanne, Switzerland
Alexandros.Stamatakis@epfl.ch

[3] Department of Computer Science
College of William and Mary
and Greek Armed Forces
Division of Research and Informatics
cda@cs.wm.edu

Abstract

Computational phylogeny is a challenging application even for the most powerful supercomputers. It is also an ideal candidate for benchmarking emerging multiprocessor architectures, because it exhibits fine- and coarse-grain parallelism at multiple levels. 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 Cell Broadband Engine, a heterogeneous multi-core processor with SIMD accelerators which was initially marketed for set-top boxes, is currently being deployed on supercomputers and high-end server architectures. We present both conventional and unconventional, Cell-specific optimizations for RAxML's search algorithm on a real Cell multiprocessor. While exploring these optimizations, we present solutions to problems related to floating point code execution, complex 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 the 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 an efficient parallel algorithm for phylogenetic tree inference, 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 [26]. RAxML uses an embarrassingly parallel master-worker model for non-parametric bootstrapping and multiple inference on distinct, reasonable randomized starting trees (more specifically, random stepwise addition sequence Maximum Parsimony trees [27]), in order to search for the best-known ML tree. RAxML's master-worker model is implemented using MPI. RAxML-VI-HPC has been further parallelized with OpenMP, to exploit loop-level parallelism in the likelihood functions. The shared-memory parallelization of the algorithm scales well and has good cache locality with inputs comprising large multi-gene alignments [28].

In this paper, we present the porting, optimization and evaluation of RAxML on a Cell multiprocessor. The Cell Broadband Engine [11] (or Cell BE), has been developed jointly by Sony, Toshiba, and IBM. Although originally intended as a processor for Sony PlayStation3, Cell is a fully operational general-purpose microprocessor, which at the same time offers a rich palette of thread-level and data-level parallelization options to the programmer. Cell has eight computation acceleration cores, named Synergistic Processing Elements (SPEs), each equipped with a vector execution unit and a vector ISA. The chip also includes an SMT PowerPC processor, named the Power Processing Element (PPE), which runs Linux and operates both as a standalone general-purpose microprocessor, and as a scheduler of computation on the SPEs. The memory and interconnection network architecture of Cell achieve a maximum on-chip data transfer bandwidth of over 200 Gigabytes/s. The PPE, SPEs and interconnect are all packaged on a single, thumb-size die, operating within 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 [30]). Power consumption on Cell is comparable to that of mobile processors [30].

1-4244-0910-1/07/$20.00

An exploration of programming models, runtime environments, compiler support and application development on Cell is highly relevant. The Cell is the processor of choice for Roadrunner, the machine which is anticipated as the first system to sustain a Petaflop, by the end of 2008[1] .

This paper makes four primary contributions:

- We present a detailed empirical optimization process of RAxML on Cell. We use a real Cell multiprocessor for this study. We show that although RAxML is seemingly an ideal target for Cell due to its inherent multilevel parallelism, optimizing RAxML on Cell is a non-trivial exercise. Moreover, we find it highly unlikely to achieve the desired level of optimization automatically, without additional support from the runtime environment.
- We quantify Cell-specific code optimizations and assess their impact, using RAxML. We find that merely exposing multi-level parallelism is insufficient for high performance. Both conventional and unconventional optimizations are important to accelerate program execution. The conventional optimizations applied 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 of the bulk of the computation on SPEs. Unconventional optimizations include the vectorization of conditional statements, asynchronous communication through direct SPE memory accesses, and interleaved event-driven scheduling of tasks across SPEs. We find, somewhat surprisingly, that unconventional optimizations such as the vectorization of conditionals, can have a larger impact on performance than conventional optimizations.
- We find that multi-level parallelization on Cell is both feasible and necessary, however its exploitation is an elaborate process. 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 SPEs) being more beneficial for realistic workloads and three layers of parallelism (task-level parallelism across SPEs, loop-level parallelization across SPEs and vectorization within SPEs), being beneficial for workloads with a low degree of task-level parallelism available per processor.
- 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 Cell prototype. The results demonstrate the superiority of Cell as a processor for high-end computing.

[1] See http://www.hpcwire.com/hpc/893353.html and several other articles in the popular press.

The rest of this paper is organized as follows: Section 2 summarizes related work on programming support for Cell 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 architectures. Section 4 outlines the Cell architecture. Section 5 presents step by step the RAxML porting and optimization process, along with experimental results for each step of the process. Section 6 compares Cell with the IBM Power5 and the Intel Xeon HT. Section 7 concludes the paper.

2 Related Work

Since the popularization of Cell for general-purpose computing tasks, 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 [31] present an analytical framework to predict performance of code written for Cell. They exercise their model using small linear algebra kernels and, driven by their observations, they propose microarchitectural extensions to improve double-precision floating point performance.

Eichenberger et. al [9] present 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. They also present 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 optimization.

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 [26], shared-memory multiprocessors [28] 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 [29].

3 RAxML-VI-HPC

RAxML-VI-HPC (v2.2.0) (Randomized Axelerated Maximum Likelihood version VI for High Performance Computing) [26] 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).

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. For example, for $n = 50$ organisms there exist $2.84 * 10^{76}$ alternative trees. The number is comparable to the 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 memory- and floating point-intensive, therefore 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 [32] and RAxML [27, 26].

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 [26] on real world datasets with $\geq$ 1,000 sequences reveals that the algorithm 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. In addition, the maximum likelihood calculations have been parallelized with OpenMP [28] to exploit shared-memory multiprocessors through fine-grained loop-level parallelism. RAxML has also been ported to a GPU (Graphics Processing Unit) [5].

The MPI version of RAxML exploits the 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 [27].

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 Cell BE integrates a multithreaded PowerPC core, called the Power Processing element (PPE), with eight computation acceleration cores, called the Synergistic Processing Elements (SPEs) [10]. The SPEs are connected around a ring-structured network called the Element Interconnect Bus (EIB). The PPE is also connected on the 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 high-end computing engines of Cell. Each SPE has 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. Each SPU can directly load instructions and data only from its local storage, and can directly store data only to its local storage. Access to the local storage of remote SPEs, as well as global RAM access is possible via DMA requests. 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 the SPU can issue two double-precision floating point operations 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. With single-precision FP arithmetic, the Cell BE is capable of a peak performance of 230.4 GFLOPS [6].

The SPE can access RAM and the local storages of remote SPEs through direct memory access (DMA) requests. The DMA transfers are handled by the MFC. 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 are 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. It is organized as a 4-ring structure and can transmit 96 bytes per cycle, thus achieving a peak bandwidth of 204.8 Gigabytes/second. The EIB can also support more than 100 outstanding DMA requests.

5 Porting and Optimizing RAxML on Cell

We ported RAxML to Cell in four steps: (i) we ported the MPI code on the PPE; (ii) we offloaded the most time-consuming parts of each MPI process on the SPEs; (iii) we optimized the SPE code using the vectorization, customization of mathematical library functions through the Cell SDK, vectorization of control statements coupled with a specialized casting transformation, overlapping of computation and communication (double buffering) and other communication optimizations; (iv) lastly, we implemented multi-level parallelization schemes across and within SPEs in selected cases, as well as a scheduler for effective simultaneous exploitation of task, loop, and SIMD parallelism. We outline these optimizations in the following sections.

The results reported in this section are obtained from a real dual-Cell multiprocessor, located at the Barcelona Supercomputing Center (http://www.bsc.es). The Cell processors on this platform run at 3.2 GHz and the system has 512 MB of XDR RAM. The PPEs have a 32 KB L1 instruction cache, a 32 KB L1 data cache, and a 512 KB unified L2 cache. The system runs Linux Fedora Core 5 (with kernel version 2.6.16), including Cell-specific kernel patches. We compiled our code using Toolchain 4.0.2.

5.1 Porting MPI Code

Mapping MPI processes to PPE threads using a one-to-one scheme on the Cell may underutilize SPEs, if the PPE threads do not expose enough task-level parallelism to offload to SPEs. To address this problem, we introduced both loop-level parallelization of tasks across SPEs and an event-driven scheduler which multiplexes more than two MPI processes on the PPE, so that more sources of task-level parallelism are made available for offloading computation on SPEs.

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 the profiling and 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, 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. Conse-

(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.**

quently, the first candidate for offloading is newview(). Although makenewz() and evaluate() are both taking a smaller portion of the execution time than newview(), offloading these two functions results in 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 reduces significantly 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 thread spawning and joining, threads remain bound on SPEs and busy-wait for the PPE signal, before starting to execute a function.

5.2.1 Optimizing Off-Loaded Functions

The discussion in this Section refers to function newview(), which is the most computationally expensive in 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 (bootstraps) performed.

As shown in Table 1, merely offloading newview() causes performance degradation. We profiled the new version of the code in order to get a better understanding of the major bottlenecks. Inside newview(), we identified 4 parts where the function spends almost its entire lifetime: The first part includes math library functions such as exp() and 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 checks are used in every ML implementation). The third time-consuming part involves DMA transfers. The fourth includes the loops that perform the actual likelihood vector calculation. In the next few sections we describe the techniques used to optimize the aforementioned parts in newview(). The same techniques were applied to the other offloaded functions.

5.2.2 Mathematical Library Functions

The code in newview() executes on average 25,554 floating point operations per invocation, for the 42_SC input data set. 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 accounts for 50% of the execution time in newview(). We replaced the PPC math library function exp() with the exp() function provided by 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. This optimized reduces execution time by 37%–41% in the tested cases.

5.2.3 Vectorizing Conditional Statements

RAxML always invokes newview() at an inner node of the tree (p) which is at the root of a subtree. The main computational kernel in 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 loop in newview() can be simplified. This optimization leads to significant performance improvements [26]. To activate the optimization, we use four 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 four execution paths in newview() leads to a distinct—highly optimized—version of the loop which per-

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 the code uses the exp() **function from the SDK library. The input file is 42_SC.**

forms the actual likelihood vector calculations. Each iteration of this loop executes the previously mentioned 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 execution time is spent in this particular conditional statement. Furthermore, almost all the time is spent in checking the condition, while negligible time is spent in the body of code in the fall-through part of the conditional statement. The problematic conditional statement is shown below. The symbol 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 the 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 made the observation 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" [16]. We used this property to replace floating point comparison with 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. With spu_and() we always set the left most bit of a floating point number to one. After computing the absolute values of all the operands involved in the if() statement, we cast each operand to an unsigned long long and perform the comparison.

Following optimization of the offending conditional statement, its contribution to execution time in newview() comes down to 6%, as opposed to 45% before optimization. The total execution time (Table 3) improves by 19%–21%.

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 a batch of elements at a time. 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.

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.

In the original code where SPEs wait for all DMA transfers, the idle time accounts for 11.4% of execution time of newview(). We eliminated the 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 data transfer size (set to 2 KBytes) is shown in Table 4.

5.2.5 Vectorization

All calculations in newview() are enclosed in two 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 second loop has 228 iterations and

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 with computation. The input file is 42_SC.

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 following vectorization. The input file is 42_SC.

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 following the optimization of communication to use direct memory-to-memory transfers. The input file is 42_SC.

executes 44 double precision floating point operations per iteration. Each SPE on the Cell is capable of exploiting data parallelism via vectorization. The SPE vector registers can store two double precision floating point elements. We vectorized the two loops in newview() using these registers. We present a simplified explanation of our vectorization strategy due to space considerations. As an example, we are using the second, larger loop. The first loop is optimized with the same method.

Figure 1 illustrates the loop (showing a few selected instructions which dominate execution time in the loop). The variables x1->a, x1->c, x1->g, and 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 vectorization 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 1 is the one that achieved the best performance in our tests. 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 drops from 36 to 24 for the first loop, and from 44 to 22 for the second loop. Vectorization adds 25 instructions for creating vectors.

Without vectorization, newview() spends 19.57 seconds (or 69.4% of its execution time) in the two loops. Following vectorization, the time spent in loops drops to 11.48 seconds, or 57% of the execution time of newview(). Table 5 shows execution times following vectorization.

5.2.6 PPE-SPE Communication

Although newview() accounts for most of the execution time, its granularity is fine and its contribution to execution time is attributed to the large number of invocations. For the 42_SC input, newview() 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, it sends the result back to the PPE.

In an early implementation of RAxML, we used mailboxes to implement the communication between the PPE and SPEs. We observed that PPE-SPE communication can be significantly improved if it is performed through main memory and SPE local storage instead of mailboxes. Using memory-to-memory communication improves execution time by 2%–11%. Table 6 shows RAxML 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.

5.2.7 Increasing the Coverage of Offloading

In addition to newview(), we offloaded makenewz() and evaluate(). All three offloaded functions were packaged in a single code module loaded on the SPEs. The advantage of using a single module is that it can be loaded to the local storage once when an SPE thread is created and remain pinned in local storage for the rest of the execution. Therefore, the cost of loading the code on SPEs is amortized and communication between the PPE and SPEs is reduced. For example, when newview() is called by makenewz() or evaluate(), there is no need for any PPE-SPE communication, since all functions already reside in SPE local storage.

Offloading all three critical functions improves performance by a further 31%–38%. A more important implication is that after offloading and optimization of all three functions, the RAxML code split between the PPE and one SPE becomes actually faster than the sequential code executed exclusively on the PPE, by as much as 25%. Function offloading is another optimization which scales with parallelism. When more than one MPI processes are used and more than one bootstraps are offloaded to SPEs by each process, the gains from offloading rise to 47%. Table 7 illustrates execution times after fill function offloading.

```
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 1. The second loop 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.**

5.3 Multilevel 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 processor and each MPI process executes one bootstrap, two MPI processes can use only two out of eight SPEs for parallel execution. We considered two programming models in order to exploit the rest of the SPEs on Cell. The first is loop-level parallelization (LLP) within a bootstrap, coupled with loop distribution across SPEs. The second 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 bootstraps for SPEs. We implemented both parallelization methods and observed that there is no single method that performs the best in all cases [4]. Consequently, we also implemented a dynamic parallelization scheme, named MGPS (Multi-grain Parallelism Scheduling) [4], which combines the LLP and EDTLP models.

In the MGPS model, the scheduler decides on-the-fly which parallelization model (EDTLP, LLP, or both) the application should use. The parallelization model may also change at runtime, depending on the application workload and more specifically the recent history of SPE utilization by off-loaded tasks [4].

Using the MGPS model we were able to further reduce the execution time of RAxML, as shown in Table 8. 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 demonstrate the impact of these schedulers on RAxML. The schedulers reduce execution time of one bootstrap by 36%, due to loop-level parallelization, and up to 63% with more bootstraps, due to simultaneous exploitation of task-level and loop-level parallelism.

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.**

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

We compare the performance of the Cell implementation of RAxML to the MPI implementation of RAxML on two multiprocessors:

- A dual-processor system, with 32-bit Intel Pentium 4 Xeon processors with Hyper-threading technology (2-way SMT), running at 2GHz, with 8KB L1-D cache, 512KB L2 cache, and 1MB L3 cache.
- A single-processor system, with one 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.

We use 42_SC as the input data set. Figure 2 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 threads. Therefore, RAxML is executed with four MPI processes on the Power5. On the Intel multiprocessor, RAxML is executed with four MPI processes, on two Intel processors with Hyperthreading activated on each processor.

The Cell clearly outperforms the Intel Xeon by a large margin, more than a factor of four on a one-to-one comparison, and more than a factor of two if one Cell is compared against two Xeons processors on the same data set. One Cell performs also 9%-10% better than one IBM Power5. Although the margin of difference between Cell and Power5 seems small, Cell has some important advantages over a general-purpose high-end processor such as the Power5. The Cell is dramatically more cost-effective and more power-efficient than the Power5 [30, 20].

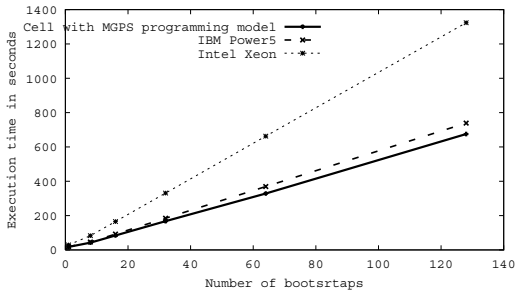

Figure 2. RAxML performance on different multiprocessors with multithreaded and multicore components: Intel Xeon HT, IBM Power5 and Cell. Execution time is plotted against the number of bootstraps.

7 Conclusions

We presented the porting, optimization and evaluation of RAxML (Randomized Axelerated Maximum Likelihood), a challenging application from the domain of computational phylogenetics, on the Cell Broadband Engine. We explored a total of seven Cell-specific optimizations and the performance implications of these optimizations: I) Offloading the bulk of the maximum likelihood tree calculation on the SPEs; II) Replacing expensive mathematical library functions with Cell-specific numerical implementations; III) Casting and vectorization of expensive conditional statements involving multiple, hard to predict conditions; IV) Double buffering for overlapping completely DMA transfers with computation; V) Vectorization of the core of the floating point computation; VI) Optimization of PPE-SPE communication using direct memory-to-memory transfers; VII) Dynamic multi-level parallelization, achieved by oversubscribing the PPE and/or exploiting loop-level parallelism when the task-level parallelism exposed by MPI processes leaves SPEs unused.

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 performance achieved by leading current multicore processors.

In future work we plan to experiment with different sizes of the offloaded functions, especially those functions with an object code size that may exceed the size of the local storage. In RAxML we did not experience this problem, since the total size of all offloaded functions occupies about 50% of the local storage. For large data sets, a more elaborate code management methodology, such as executing non-leaf procedures on the PPE and off-loading leaf procedures on the SPEs, or stack spilling and retrieval from the SPEs, is needed. We also intend to incorporate different scheduling policies in our current runtime system. The need for customized, application-specific scheduling policies emerges together with the porting of more real-world parallel applications on the Cell.

Acknowledgments

This research is supported by the National Science Foundation (Grants CCR-0346867, ACI-0312980, CNS-0521381), the U.S. Department of Energy (Grant DE-FG02-06ER25751), the Swiss Confederation Funding, the Barcelona Supercomputing Center, which generously granted us access to their Cell multiprocessor, and 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] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic Multigrain Parallelization on the Cell Broadband Engine. *2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 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] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation. *IBM developerWorks*, Nov 2005.

[7] B. Chor and T. 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] M. Kistler, M. Perrone, and F. 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] B. Q. Minh, L. S. Vinh, A. von Haeseler, and H. 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. RAxML-VI-HPC: Maximum Likelihood-Based Phylogenetic Analyses With Thousands of Taxa and Mixed Models. *Bioinformatics*, page btl446, 2006.

[27] 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.

[28] 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.

[29] 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.

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

[31] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The Potentinal of the Cell Processor for Scientific Computing. *ACM International Conference on Computing Frontiers*, May 3-6 2006.

[32] D. 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.