

Exploring Programming Models and Optimizations for the Cell Broadband Engine using RAxML

Filip Blagojevic and Dimitrios S. Nikolopoulos
Department of Computer Science
College of William and Mary
P.O. Box 8795, Williamsburg, VA 23187-8795

Abstract

Originally developed as a gaming processor for Sony PlayStation3, the Cell Broadband Engine opens new opportunities for running computationally intensive scientific applications more efficiently, thanks to characteristics such as multigrain task-level and data-level parallel execution and vast on-chip memory bandwidth. In the ideal case, the Cell is capable of achieving significant performance improvements over conventional processors. However, the potential of the Cell is unclear when the processor is used for applications that are not necessarily conforming to its architectural characteristics. Furthermore, the question of what is the best programming model for a processor like Cell remains open, with too many programming models and paradigms proposed, yet too few evaluated empirically or experimentally. In this work we present the port and optimization of RAxML, an application that computes large phylogenetic trees, on a real blade with Cell processors. We investigate two programming models that derive partially from the dominant programming models of conventional parallel machines, namely MPI and OpenMP, as well as an extensive set of Cell-specific optimizations. Using multilevel parallelization and several optimizations we have been able to improve the execution time of RAxML on the Cell by a factor of 5, a satisfactory result given that RAxML is an application with dynamically allocated data structures, complex control flow and extensive pointer arithmetic, all factors that present challenges for parallelization beyond the simple master-worker scheme. We also find that the Cell performs comparably or outperforms leading multicore and multithreaded microprocessors, such as the IBM Power5 and the Intel Xeon with Hyperthreading technology.

1 Introduction

The Cell Broadband Engine [8] has been developed jointly by Sony, Toshiba, and IBM. Although primarily intended as a processor for Sony PlayStation3, the Cell can theoretically support a broad range of applications. In particular, the Cell appears to be well suited for scientific data-intensive applications with high demands for memory bandwidth. The Cell Broadband Engine is a heterogeneous multiprocessor with nine execution cores: one SMT Power Processing Element (PPE) based on the PPC architecture and eight single-threaded Synergistic Processor Elements (SPEs). Among other interesting features, SPEs integrate powerful SIMD execution capabilities.

According to the specifications [1], the Cell is capable of achieving significant performance improvement over conventional CPUs, reaching peak performance levels of over 200 Gflops. However, due to its unconventional architecture, developing applications that can exploit the architectural strengths of the Cell, most notably the multiple levels of parallelism and its memory bandwidth, is an arduous task. One of the main difficulties is the management of the local storage of the SPEs by software. Another difficulty is the distribution of work among SPEs, which can be done either at the granularity of complete functions, in the spirit of task-level parallelization, or at a finer granularity by parallelizing execution constructs within off-loaded functions.

At the time of writing this paper, only a few computational kernels, such as matrix multiplication and FFT, and one well-known DOE code (Sweep3D), have been ported and optimized for the Cell. These applications have been parallelized with different models, including hand parallelization and automated compiler-driven parallelization. The results produced from these experiments are promising. The potential of the Cell to execute more efficiently complex applications with irregularities in their access patterns, dynamic data structures, pointer arithmetic, complex control flow, fine granularity of computation, and extensive double-precision floating point arithmetic, all of which are mismatches to Cell's architectural strengths, is an interesting, open question.

In this work, we study RAxML [12, 11, 13] (Randomized Accelerated Maximum Likelihood) on a real Cell blade server. RAxML is a computational biology application used to determine relationships between different species.

The application is extremely computationally intensive, and the amount of computation grows exponentially with the number of species. The execution time for certain inputs can reach several days. RAxML is parallelized with MPI using a master-worker paradigm, and contrary to the simple kernels already ported to Cell, it is not an ideal target for parallelization on the broadband engine for the following reasons:

- Due to inter-process communication and memory requirements, MPI processes can not be completely off-loaded on the SPEs. A significant part of the application still needs to be executed on the PPE.
- RAxML does not have temporal locality. Consequently, data needs to be continuously streamed from and to the local storages of the SPEs.
- All memory used in the code is allocated dynamically and the runtime system can not know far in advance the addresses of the data that needs to be fetched in the local storages of the SPEs.
- Many computational parts of the application have to be significantly restructured in order to be vectorized. In particular, the code is practically impossible to vectorize automatically with a compiler.
- The application has an additional hidden level of parallelism within the MPI nodes, which can be exploited by parallelizing loops. This additional level of parallelism needs to be exploited to effectively utilize all the SPEs on the Cell, however the granularity of the computation at this level is very small (in the order of a few tens of microseconds).

We explore two programming models for parallelizing RAxML on the Cell, namely function off-loading and function offloading combined with parallelization of loops using work-sharing across multiple SPEs. Both programming models use vectorization and implement up to four levels of parallelism (task-level parallelism within the PPE, task-level and loop-level parallelism across the SPEs and vectorization within the SPEs). We also implemented several Cell-specific optimizations. The programming models and the optimizations we employed improved the performance of RAxML when executed on Cell by more than a factor of 5. Our experiments show that the performance of RAxML when executed on Cell, is better than the performance of a Power5, a dual-core processor with two-way SMT cores, when the two cores are used in non-SMT mode. Power5 has a slight edge when SMT is activated in its cores. We also find that the Cell outperforms dramatically the Intel Xeon processor with Hyperthreading technology.

2 Related Work

Kistler et. al [10] presented a first comprehensive performance evaluation of the Cell's on-chip interconnection network. They conducted a series of experiments to estimate DMA performance on the Cell. For this purpose they developed a set of microbenchmarks written in C. They reported the latencies of DMA requests, the DMA bandwidth achieved between local storage and main memory and the maximum number of outstanding DMA requests. They also investigated the system behavior under different patterns of communication between local storages and main memory.

Williams et. al [14] developed an analytical framework to predict performance on the Cell. In order to test their model, they used kernels like dense matrix multiplication, sparse matrix vector multiplication, stencil computations, and 1D/2D FFTs. In addition, they proposed micro-architectural modifications that would increase the performance of the Cell on codes with double-precision floating point arithmetic.

Eichenberger et. al [6] presented several compiler techniques targeting automatic generation of highly optimized code for the Cell. They tried to exploit the multiple levels of parallelism available on the Cell. The techniques they presented include compiler-assisted memory alignment, branch prediction, SIMD parallelization, and OpenMP task level parallelization. They also designed a compiler-controlled software cache. Unfortunately, such automated parallelization capabilities are hard to deploy in codes such as RAxML due to the use of dynamically allocated data structures and extensive pointer arithmetic.

3 Cell Overview

The Cell is composed of a Power Processing element (PPE) and eight Synergistic Processing Elements, or SPEs [7]. These elements are connected with an on-chip element interconnect bus (EIB).

The PPE is a 64-bit, multi-threaded Power Architecture [2] processor, with Vector/SIMD Multimedia extensions [3] and two levels of on-chip cache. The size of the L1 instruction and data caches is 32 KB, while the size of the second level cache is 512 KB. On the Cell we used for this research, the PPE is a 2-way SMT with modest performance and runs Linux.

The SPEs are the primary high-performance computing engines on the Cell. Each SPE is a 128-bit processor with two major components: a Synergistic Processing 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 a 256 KB 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, however only one operation can be issued per SPU cycle. Double precision floating point operations are partially pipelined and two double-precision floating point operations can be issued every six cycles. With eight SPUs and fully pipelined double-precision floating-point support in the PPE's VMX, the Cell BE is capable of a peak performance of 21.0 Gflops for double-precision floating-point arithmetic, and 230.4 Gflops for single-precision floating point arithmetic [5].

The SPE can access main memory through 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 up to 16 KB. To transfer large amounts of data the SPE uses DMA lists. A DMA list can hold up to 2,048 DMA transfers, each for up to 16 KB. The MFC supports DMA transfer sizes that are 1, 2, 4, 8 or multiples of 16 bytes.

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

4 RAxML

RAxML (Randomized Accelerated Maximum Likelihood) is an application designed to derive evolutionary (phylogenetic) trees, based on the Maximum Likelihood method [9]. Phylogenetic trees represent the evolutionary connection among different organisms. Creating phylogenetic trees has many applications in biology and medicine for studying the evolution and spread of different viruses such as HIV [4]. The application is extremely computationally intensive due to the fact that the number of possible tree topologies grows exponentially with the number of organisms.

The heuristics used in RAxML belong to a class of algorithms which optimize the likelihood of a random starting tree. The starting tree is created from the DNA sequences of all the organisms. The DNA sequences are specified in an input file. RAxML searches for the best likelihood tree by performing subtree rearrangements: all possible subtrees are removed and re-inserted to the neighboring branches. If one of these new topologies improves the likelihood of the current tree, the tree is updated, and all possible subtrees are rearranged again. The rearranging process is repeated until no tree with better likelihood is found. One problem with the rearranging algorithm is that the final tree depends strongly on the starting tree. Therefore, RAxML is capable of calculating many different maximum likelihood trees, starting from random trees. This process is performed in parallel using MPI.

The parallel implementation of RAxML is based on a simple master-worker paradigm. The master initiates the optimization process by sending a starting message to the workers. Each worker performs two steps:

1. it creates a random starting tree using the input DNA sequences;
2. it calculates the best likelihood tree performing the rearranging process.

After finishing the rearranging process, the workers send the best likelihood tree to the master. The process is repeated until the code calculates a number of provably best likelihood trees set by the user.

The main part of the computation is performed by the workers, while the only task of the master is to distribute work. The communication between the master and the workers is therefore low: one message when the master sends a subtree to a worker, and one message when the master receives a rearranged tree from a worker. The size of the messages can be large, since the whole tree is sent in a single message. For example, with the input 42_SC, which contains 42 organisms where each organism is represented by a DNA sequence of 1167 nucleotides, the size of a single message is 5 KB. In our experiments we used workloads with up to 32 starting trees.

The major data structures in RAxML are pointer-based and they are allocated dynamically. This makes vectorization and parallelization of the code non-trivial. Another feature of RAxML that hinders optimization on the Cell is its fine granularity. Although the application is computationally expensive due to the exponential growth of its search space, each point in the search space is processed within tens of microseconds. Therefore, multigrain parallelization within the main tasks of RAxML can be a challenge.

5 Porting the MPI implementation of RAxML to Cell

The straightforward approach to porting RAxML on the Cell is to let the MPI processes execute on the PPE, while some, hopefully the most significant, part of each MPI process is off-loaded to an SPE. Since the PPE has only 2 hardware threads, the code can meaningfully use at most 2 MPI processes on the PPE and 2 SPEs. We will explore methods to overcome this limitation in the next section.

Our first step in the porting and optimization process was to find the parts of the program that are suitable for off-loading to the SPEs. We profiled the code using `gprofile` to identify the computationally intensive functions and off-load them to the SPEs. We ran the code on an IBM Power5 processor, using `42_SC` as the input. We found that 98.4% of the execution time is spent in three functions: 76.8% in `newview()`, 19.2% in `makenewz()`, and 2.4% in `evaluate()`.

5.1 Function Off-loading

To off-load a function from within the MPI node to an SPE, we spawn an SPE thread at the beginning of each MPI process. The thread executes the function upon receiving a signal from the PPE and returns the result back to the PPE upon completion. To avoid excessive overhead, SPE threads are persistent and they busy-wait until they receive signals from the PPE.

To keep the implementation simple, the call to each off-loaded function is retained with the original signature on the PPE. We replaced the original function body with the communication code needed to transfer trees from the PPE to the SPE. Whenever an off-loaded function is called, the PPE sends a signal to the SPE thread and waits for the SPE thread to complete the function and return the result. While waiting for the SPE thread to finish, the PPE busy waits.

The rest of the discussion refers to function `newview()`, which is the most computationally expensive of the code. Even though around 77% of the total time is spent in `newview()`, the per-invocation cost of this function is very low. On an IBM Power 5, `newview()` is invoked 260,189 times for the `42_SC` input file, and the average execution time of a single invocation is 52 μ s. Therefore, care must be taken to balance the cost of off-loading `newview()` with adequate optimizations. Furthermore, vectorizing or parallelizing parts of `newview()` on an SPE needs to be done with care to avoid excessive overhead. On the other hand since the waiting period for the PPE is small, busy waiting is clearly a better option than putting the MPI process to sleep and waking it up to join the SPE thread.

Table 1 shows the execution times of RAxML before and after `newview()` is off-loaded. All execution times reported in this and following tables are arithmetic means from ten runs. The variance in all reported results is very small (at most ± 0.3 seconds). The first column shows the number of workers used in the experiment and the amount of work done. We use problem size scaling in the parallel executions of the application: N ML trees signifies that N maximum likelihood trees are calculated. For the scaled experiments we run 4 MPI processes across 2 Cell processors hosted on the same blade server. The computation of N ML trees on one worker takes exactly N times longer than the computation of 1 ML tree. Therefore, with 4 workers, 8 ML trees ideally would need twice as much time as 1 ML tree on one worker, or around 84 seconds. The observed execution time (123 seconds) is higher due to the fact that the two Cells use SMT PPEs, and although parallel speedup across the Cells is good, parallel speedup within each Cell's SMT PPE is far from ideal.

(a)	1 worker, 1 ML tree	41.9 s	(b)	1 worker, 1 ML tree	137 s
	4 workers, 8 ML trees	123 s		4 workers, 8 ML trees	286.3 s
	4 workers, 16 ML trees	248 s		4 workers, 16 ML trees	571.2 s
	4 workers, 32 ML trees	485 s		4 workers, 32 ML trees	1140 s

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 off-loaded to one SPE.

The first result in Table 1 is disappointing. Despite off-loading a major part of the code to the SPEs, the code experiences significant performance degradation compared to the base case where the entire code is executed on the PPE, obviously due to the absence of SPE-specific optimization. Using the decremented register to measure the time spent in the SPE thread, we identified four code regions where the `newview()` spends its cycles:

- math library functions such as `exp()` and `log()` that are used in the function;
- several large, nested `if()` statements;
- DMA transfers;

- the major likelihood tree calculations enclosed in two loops.

In the next sections we describe the techniques used to optimize `newview()`. Similar techniques were applied to the other off-loaded functions.

5.2 Math functions

On average, `newview()` executes 25,554 flops during a single invocation. 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 that 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. Replacing the `exp()` function yields a dramatic performance improvement and almost halves execution time.

1 worker, 1 ML tree	76 s
4 workers, 8 ML trees	166 s
4 workers, 16 ML trees	333.3 s
4 workers, 32 ML trees	633.3 s

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

5.3 Vectorizing if() statements

The main computational kernel of `newview()` has a `switch` statement which selects one out of four paths of execution. Each path leads to a large loop with a conditional statement in the end of each iteration. Mis-predicted branches in the code that implements this statement incur a penalty of approximately 20 SPE cycles. We profiled `newview()` after replacing the math library functions and found that 45% of the function execution time is spent in the one conditional statement at the end of the body of the loop. Furthermore, almost all the time is spent in checking the condition, while negligible time is spent in the body of the conditional statement. The offending conditional statement is shown in Figure 1, where `minlikelihood` is a constant.

```

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

```

FIGURE 1: A single conditional statement that takes 45% of the execution time of `newview()`.

This statement is a challenge for a branch predictor, since it integrates 8 conditions, one for each of the four `ABS()` macros and four comparisons against the minimum likelihood. Using Cell intrinsics, we managed to remove almost all condition checking by vectorizing the offending statement. We transformed the conditional statement of Figure 1 into a vector operation that contains only one condition checking. Our transformation is based on the usage of the vector intrinsic `spu_cmpabsgt(v2, v1)`. This intrinsic compares the absolute value of each element of vector `v1` to the absolute value of the corresponding element of vector `v2`. If the element of `v1` is greater than the element of `v2`, the corresponding element of the resulting vector is set to one; otherwise, it is set to zero. Instead of using the large `if()` statement, the idea is to create two vectors `v1 = (x3->a, x3->c, x3->g, x3->t)`, and `v2 = (minlikelihood, minlikelihood, minlikelihood, minlikelihood)`, and apply the `spu_cmpabsgt()` intrinsic to these two vectors. A potential problem with this technique is that `spu_cmpabsgt()` operates only on single precision floating point vectors, while all the variables used in the original conditional statement are doubles. If we simply try to cast one of the doubles to a float, we will lose precision. Since we are dealing with very small values (`minlikelihood` is equal to 2^{-256}), values that are larger than `minlikelihood` can become smaller

during the casting process. To overcome this problem, we first converted the conditional statement in Figure 1 to the statement shown in Figure 2. This conversion is possible since `minlikelihood` is greater than 0.

```

double twotothe256 = 2^256;
if ( (ABS(x3->a)*twotothe256)<1 && (ABS(x3->g)*twotothe256)<1 &&
      (ABS(x3->c)*twotothe256)<1 && (ABS(x3->t)*twotothe256)<1 )
    {
        . . .
    }

```

FIGURE 2: Converted conditional statement. The condition `x<minlikelihood` is replaced with `x/minlikelihood<1`, where $x \in \{\text{ABS}(x3 \rightarrow a), \text{ABS}(x3 \rightarrow c), \text{ABS}(x3 \rightarrow g), \text{ABS}(x3 \rightarrow t)\}$.

Following the conversion, if we try to cast $\text{ABS}(x3 \rightarrow a) * 2^{256}$ from double to float we might lose some precision. However if $\text{ABS}(x3 \rightarrow a)$ is less than 2^{-256} then the casted value will be less than 1. If $\text{ABS}(x3 \rightarrow a)$ is greater than 2^{-256} then the casted value will be greater than one. The reason is the IEEE representation of floating point numbers. Therefore, the new conditional statement will have the same resulting value as the starting conditional statement. Using the above transformation we are able to create two single precision floating point vectors: $v1 = (x3 \rightarrow a * 2^{256}, x3 \rightarrow c * 2^{256}, x3 \rightarrow g * 2^{256}, x3 \rightarrow t * 2^{256})$ and $v2 = (1, 1, 1, 1)$, and apply the `spu_cmpabsgt()` intrinsic to these vectors. Actually, we apply the combination of `spu_cmpabsgt()` and `spu_gather()` to vectors $v1$ and $v2$. A short explanation of `spu_gather()` is shown in Figure 3. With this implementation, if element 0 of the resulting vector is equal to 0, our starting `if()` statement will be true. The final code is shown in Figure 3.

`spu_gather(a)` – The rightmost bit of each element of vector a is gathered, concatenated, and returned in the rightmost bits of element 0 of the resulting vector.

```

vector double r;
vector float v1;
vector float v2 = (vector float)(1,1,1,1);
vector float s[2];

r = (vector double)(x3->a,x3->c);
r = spu_mul(x3_sum_v, twotothe256_v);
v2 = spu_insert((float)spu_extract(r,0),v2,0);
v2 = spu_insert((float)spu_extract(r,1),v2,1);

r = (vector double)(x3->g,x3->t);
r = spu_mul(x3_sum_v, twotothe256_v);
v2 = spu_insert((float)spu_extract(r,0),v2,2);
v2 = spu_insert((float)spu_extract(r,1),v2,3);

s[0] = spu_cmpabsgt(v2,v1);
s[1] = spu_gather(s[0]);

if (__unlikely(spu_extract(s[1],0)==0)){
    . . .
}

```

FIGURE 3: The final version of the code that replaces the conditional statement of Figure 1.

The code after vectorization of the conditional statement takes only 6% of the execution time in `newview()`. The execution time improves significantly (by more than 17%) from the vectorization of `if()`. The new execution times

of our workloads are shown in Table 3.

1 worker, 1 ML tree	56.5 s
4 workers, 8 ML trees	130.9 s
4 workers, 16 ML trees	259.3 s
4 workers, 32 ML trees	520 s

TABLE 3: Execution time of RAXML after the expensive conditional statement is removed. The input file is 42.SC.

5.4 Double Buffering and Memory Management

The second of the two major tree calculation loops in `newview()` can execute up to 50,000 iterations. The loop operates on large arrays, with members pointing to `likelihood_vector` data structures, each of which is padded to 128 bytes, in order to facilitate memory transfers on the Cell. The arrays are allocated dynamically. Since there is no limit on the size of these arrays, the arrays can not be stored entirely in the local storages of the SPEs. The SPEs can only fetch a few array elements to their local storage, execute the loop iterations that access or modify these elements, send the results back to memory, if needed, and repeat the process. Since the loop operates on three different large arrays, we allocate three buffers in the local storage of the SPEs. The size of each buffer is 2 KB, which is enough to store data for 16 loop iterations. This size was selected experimentally to optimize DMA transfers.

If we set the SPEs to wait for the DMA transfers, the idle time accounts for 11.4% of the total execution time of `newview()`. We eliminated the DMA 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 to 2 KB is shown in Table 4. Double buffering and latency overlap improve performance by 6%.

1 worker, 1 ML tree	53.5 s
4 workers, 8 ML trees	122.6 s
4 workers, 16 ML trees	246.2 s
4 workers, 32 ML trees	488.9 s

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

Note that the space used for double buffering is much smaller than the size of the local storage. Considering that the loop executes a recursion, we opted to keep the buffers small enough, so that the recursion in the workloads we used could be executed without overflowing the local storage. The recursions in `newview()` are conditionally executed. A simple solution to the problem of local storage overflow due to stack growth is to commit the stack to main memory before the function executes a new level of the recursion. After returning from that level, the whole stack of the function can be retrieved from main memory and the function can resume. To reduce the number of memory transfers further, instead of committing the stack before each recursive call, the runtime system can wait for several recursive calls (until the memory needed for the stack reaches the limit of the local storage), and then move the accumulated stack to main memory. Implementation of these mechanisms and evaluation with large workloads is ongoing work beyond the scope of this paper.

5.5 Vectorization

Almost all tree calculations in `newview()` are executed in two loops. Although the trip count of the loops depends randomly on the input, the work per iteration is constant. The first loop executes 36 double precision floating point operations per iteration. The second loop executes 44 double precision floating point operations per iteration. Each SPE on the Cell is capable of exploiting data parallelism via vectorization, using 128-bit vector registers. We present a simplified explanation of our vectorization strategy for the dominant loops in `newview()`.

The kernel of the first loop in `newview()` is shown in Figure 4(a). In Figure 4(b) we show the same code vectorized for the SPE. The `spu_mul()` vector operation multiplies two vectors (in this case the arguments are vectors of doubles.) The `_exp_v()` operation is the vector version of the exponential calculation described in Section 5.2. After vectorization, the number of floating point instructions executed in the body of the first loop is 24. Also, there is one additional instruction for creating a vector from a scalar element. Note that due to involved pointer arithmetic

<pre>(a) for(...) { ki = *rptr++; dlc = exp (ki * lz10); dlq = exp (ki * lz11); dlt = exp (ki * lz12); *left++ = dlc * *EV++; *left++ = dlq * *EV++; *left++ = dlt * *EV++; *left++ = dlc * *EV++; *left++ = dlq * *EV++; *left++ = dlt * *EV++; . . . } </pre>	<pre>(b) vector double *left_v = (vector double*)left; vector double lz1011 = (vector double)(lz10,lz11); . . . for(...) { ki_v = spu_splats(*rptr++); dlqg = _exp_v (spu_mul(ki_v,lz1011)); dltc = _exp_v (spu_mul(ki_v,lz1210)); dlgt = _exp_v (spu_mul(ki_v,lz1112)); left_v[0] = spu_mul(dlqg,EV_v[0]); left_v[1] = spu_mul(dltc,EV_v[1]); left_v[2] = spu_mul(dlgt,EV_v[2]); . . . } </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

FIGURE 4: The body of the first loop in `newview()`: (a) Non-vectorized code, (b) Vectorized code.

on dynamically allocated data structures, automatic vectorization of this code would be extremely challenging for a compiler.

Other loops in the off-loaded function are vectorized in a similar way. Figure 5 presents an example of one of the larger loops (showing a few instructions that dominate the body of the loop). The variables `x1->a`, `x1->c`, `x1->g`, `x1->t`, belong to the same structure and occupy contiguous memory locations. Only three of these variables are multiplied by the elements of the array `left`. Once again, automatic vectorization is very difficult, since besides pointer analysis, the code requires more vector creating instructions such as `spu_splats()`. Obviously, there are many different possibilities when vectorizing this code. The vectorization described in Figure 5(b) is the one that gave us the best speedup so far. The new vector instruction we used in this loop is `spu_madd()` which implements a fused multiply-add operation. After vectorizing the second loop, the number of floating point instructions executed in the body of this loop is 22. There are 24 additional instructions for creating vectors.

Without vectorization, `newview()` spends 19.57 seconds (or 69.4% of its total execution time) in loops. After vectorization, the time spent in loops drops to 11.48 seconds, and accounts for 57% of its execution time. The total execution time of the application decreased by 10–16% by using vectorization. Since the application uses double precision floating point arithmetic, a maximum improvement factor of 2 is expected by the SIMD execution engine of the SPEs. The improvements from vectorization are far more limited due to the fine granularity of the computations and the overhead of creating and managing vectors. The improved total execution times after vectorization are shown in Table 5.

1 worker, 1 ML tree	45 s
4 workers, 8 ML trees	111 s
4 workers, 16 ML trees	217.9 s
4 workers, 32 ML trees	435.2 s

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

5.6 Off-loading Other Functions

After off-loading and optimizing `newview()`, we continued with off-loading the next two most expensive functions: `makenewz()` and `evaluate()`. The off-loaded functions are written to the same SPE file as `newview()`. After

<pre>(a) 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++; . . . } </pre>	<pre>(b) 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]); . . . } </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

FIGURE 5: Example of one of the large loops in `newview()`: (a) Non-vectorized code, (b) Vectorized code.

expanding the existing SPE file, the size of the code segment in the SPE’s local storage rises to 100KB. We observed that the larger code segment did not influence the SPE program execution. There was still enough memory left for the stack and the heap. Furthermore, data could be pipelined between the functions running on the SPEs without committing results to main memory. Following the off-loading and optimization of all major functions, the code using one thread on the PPE and one SPE is about 29% faster than the code that uses only one thread on the PPE. The parallel code with 4 MPI processes split between the PPE and the SPEs is about 40% faster than the code that does not off-load computation on the SPEs. The execution times are shown in Table 6.

1 worker, 1 ML tree	32 s
4 workers, 8 ML trees	68.3 s
4 workers, 16 ML trees	135.15 s
4 workers, 32 ML trees	265.8 s

TABLE 6: Execution time of RAxML after off-loading three functions: `newview()`, `makenezz()` and `evaluate()`. The input file is 42_SC.

6 Multi-level Parallelization: Work-Sharing between SPEs

The function off-loading model described so far underutilizes the SPEs, since it uses a one-to-one mapping between MPI processes and SPEs. One way to overcome this problem is to overload the PPE by more MPI processes than the number of hardware threads. Unfortunately, multiplexing of hardware threads between MPI processes introduces contention and renders the PPE a major bottleneck. An alternative to multiplexing on the PPE is to further divide the off-loaded computation among SPEs.

To achieve this effect, we parallelized the off-loaded functions using work-sharing constructs similar to those encountered in OpenMP. Other than a single loop executed in `newview()` that was very short to be profitably parallelized, we were able to parallelize all loops executed in the off-loaded functions by splitting their iterations among SPEs and obtain measurable speedup.

The basic work-sharing scheme we used is presented in Figure 6, for the case where the loop is work-shared between two SPEs. Before the loop is executed, SPE1 sends a signal to SPE2. After sending the signal, SPE1 executes its assigned fraction of the loop. At the same time, SPE2 fetches all data necessary for the execution of its part of the loop. After completing its computation, SPE2 can either send the results back to SPE1 or commit the results directly to main memory, depending on whether the loop modifies array elements or updates a single value in a reduction.

After accumulating all optimizations, we find that in the sequential case (execution with one worker) the application spends 85.3% of its time in the off-loaded functions. Table 7 summarizes the execution times of RAxML after applying

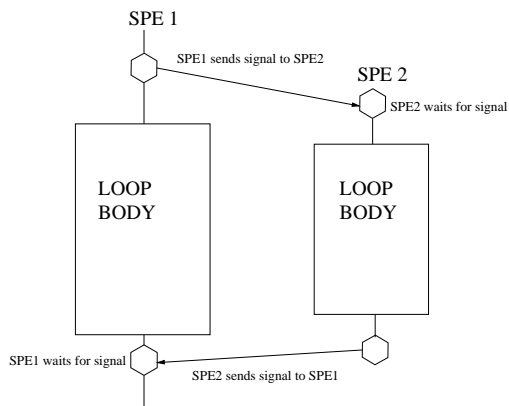


FIGURE 6: Parallelizing a loop between two SPEs using a work-sharing model.

all optimizations, including function off-loading and multi-level parallelization with work-sharing in parallel loops. We present results with work-sharing between two SPEs for each loop, using a total of 8 SPEs across 2 Cell processors on our experimental platform. The results using all 16 SPEs of the 2 Cells were unstable, and although the averages across many runs were consistent with the observed trends (i.e. mean execution time scaled by a factor of 2 in the 4 workers, 64 ML trees case) we decided to omit them until we discover the reason for the occasional erratic behavior.

1 worker, 1 ML tree	23.9 s
4 workers, 8 ML trees	51.9 s
4 workers, 16 ML trees	102.4 s
4 workers, 32 ML trees	205 s

TABLE 7: Execution time of RAxML after using work-sharing to parallelize large loops. The input file is 42_SC.

7 Performance Comparison with Other Platforms

We compared the performance of the Cell to other architectures using RAxML. Besides Cell, we evaluated the MPI version of RAxML on the following platforms:

- A Dell PowerEdge 6650 multiprocessor with Intel Xeon processors using Hyperthreading technology (2-way SMT). The processors run at 2GHz and they have 8KB L1-D caches, 12 KB instruction trace caches, 512KB unified L2 caches, and 1 MB unified L3 caches.
- An OpenPower 720 multiprocessor with 64-bit Power5 processors. The Power5 is a dual-core processor with a 2-way SMT architecture in each code. The processors run at 1.6 GHz and they have 32KB L1-D caches, 32KB L1-I caches, 1.92 MB unified L2 caches, and 36 MB unified L3 caches.

The Cell machine we used for this work is located at the Barcelona Supercomputing Center. The system is a dual Cell blade. The configuration of each Cell processor is given in Section 3. The operating system running on the blade is Fedora Core 5, and the Linux kernel version is 2.6.16, with some Cell-specific kernel patches. We used the Cell tool-chain version 2.3 to compile RAxML.

For all experiments, we used 42_SC as an input file. Figure 7 shows execution time against number of ML trees created. We used one IBM Power5 processor, running two MPI processes on separate cores (Figure 7(a)) and four MPI processes by activating both dual-core and dual-SMT mode in each core (Figure 7(b)). We used one Intel Xeon processor to run two MPI processes on the two threads of the Hyperthreaded execution engine (Figure 7(a)), and two Xeon processors located on the same board to run four MPI processes (Figure 7(b)). Finally, we used one Cell to run two MPI processes for the results reported in both Figure 7(a) and Figure 7(b), i.e. we did not scale the Cell experiments to two Cells. We scale the workload by scaling the number of ML trees, however we keep the number of MPI processes (workers) constant to 2 or 4 for the IBM Power5 and the Intel Xeon and 2 for the Cell. On the Cell only,

additional co-workers are spawned to share work during loop execution on the SPEs. Note that we are only spawning two co-workers for each worker encountering a parallel loop on the Cell, therefore only four out of the eight SPEs are active on the Cell at any time. The Cell performance results can be treated as conservative. As reported earlier, we have not been able to collect stable results with all eight SPEs active on each processor and we are investigating this behavior.

In the worse case for Cell, the optimized RAxML code performs 10% slower than the IBM Power5 processor when four MPI processes are executed in dual-core dual-SMT mode on the Power5 (Figure 7(b)). We expect this margin to narrow or vanish when all eight SPEs are activated on the Cell. The Cell still outperforms the Power5 by 45% when the hardware threads are used in dual-core mode with SMT deactivated. The performance difference between the Cell and the Intel Xeon is more striking, varying from 75% up to a factor of 3.5, despite that we favor the Xeon results by using one more physical processor.

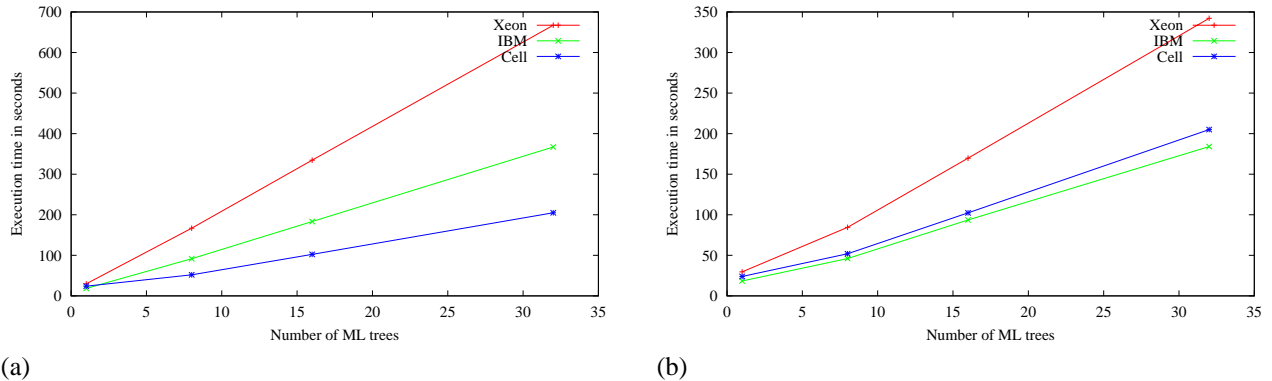


FIGURE 7: Execution time of RAxML on the IBM Power5 and the Intel Xeon with Hyperthreading technology when: (a) 2 MPI processes are executed on two cores (Power5) or two SMT threads (Xeon); (b) 4 MPI processes are executed on two cores with SMT activated (Power5), or two processors in SMT mode (Xeon). In both charts, the performance of the Cell processor is measured when running 2 MPI processes on the SMT PPE.

8 Conclusions

We presented the parallelization and optimization of RAxML (Randomized Accelerated Maximum Likelihood), an important application from the domain of computational biology, on the Cell Broadband Engine. We have explored two programming models:

- The function off-loading model: this model derives directly from MPI. MPI processes are executed on different hardware threads of the PPE, and each MPI process has three dominant functions off-loaded to an SPE. The functions off-loaded to SPEs cover over 85% of the computationally intensive part of the code.
- A hybrid programming model combining function off-loading and loop-level work-sharing.

We also applied four Cell specific optimizations on the off-loaded functions, executed on the SPEs:

- We replaced expensive mathematical functions with Cell-specific numerical implementations of the same functions.
- We vectorized expensive conditional statements involving multiple, hard to predict conditions.
- We used double buffering to overlap completely DMA transfers with computation.
- We vectorized the computation by hand to exploit the SIMD capabilities of the SPEs, where pointer arithmetic and dynamically allocated structures make vectorization a challenge for the compiler.

We have been able to improve the performance of the out-of-the-box MPI implementation of RAxML on the Cell by more than a factor of five. Our results show that the performance of RAxML on the Cell exceeds dramatically the performance of Intel SMT processors, as well as the performance of IBM Power5 processors when operated in dual-core, non-SMT mode. The IBM Power5 has only a slight advantage when the cores are operated in SMT mode, however this advantage may vanish when our implementation is scaled to use all 8 SPEs of the Cell.

Acknowledgments

This research is supported by the National Science Foundation (Grants CCR-0346867 and ACI-0312980), the U.S. Department of Energy (Grant DE-FG02-05ER2568) and equipment donations from the Barcelona Supercomputing Center and the Universitat Politecnica de Catalunya. We would like to thank Alexandros Stamatakis (the author of RAxML), Christos Antonopoulos (for helpful discussions during the initial stages of this project) and the technical staff of the Barcelona Supercomputing Center.

References

- [1] Cell broadband engine programming tutorial version 1.0; see <http://www-106.ibm.com/developerworks/eserver/library/es-archguide-v2.html>.
- [2] Power architecture version 2.02; see <http://www-106.ibm.com/developerworks/eserver/library/es-archguide-v2.html>.
- [3] PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual. <http://www-306.ibm.com/chips/techlib>.
- [4] D.A. Bader, B.M.E. Moret, and L. Vawter. Industrial Applications of High-Performance Computing for Phylogeny Reconstruction. *SPIE ITCOM: Commercial Applications for High-Performance Computing (SPIE ITCOM2001)*, August 2001.
- [5] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation; see <http://www-128.ibm.com/developerworks/power/library/pa-cellperf>.
- [6] A. E. Eichenberger et al. Optimizing Compiler for a Cell processor. *Parallel Architectures and Compilation Techniques*, September 2005.
- [7] 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.
- [8] 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.
- [9] J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol.*, 17:368–376, 1981.
- [10] 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>.
- [11] 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.
- [12] Alexandros Stamatakis, Thomas Ludwig, and Harald Meier. RAxML: A Parallel Program for Phylogenetic Tree Inference. *Proceedings of 2nd European Conference on Computational Biology (ECCB2003)*, September 2006.
- [13] Alexandros Stamatakis, Michael Ott, and Thomas Ludwig. RAxML-OMP: An Efficient Program for Phylogenetic Inference on SMPs. *PaCT*, pages 288–302, 2005.
- [14] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The Potential of the Cell Processor for Scientific Computing. *ACM International Conference on Computing Frontiers*, May 3-6 2006.