Fine-grain Parallelism using Multi-core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function

[†] Frederico Pratas [‡ *] Pedro Trancoso [≀] Alexandros Stamatakis
[† *] Leonel Sousa

[†] SiPS group, INESC-ID/IST Universidade Técnica de Lisboa, Lisbon, Portugal
{fcpp,las}@inesc-id.pt
[‡] CASPER group, Department of Computer Science, University of Cyprus, Nicosia, Cyprus
pedro@cs.ucy.ac.cy
[≀] The Exelixis Lab, Bioinformatics Unit (I12)
Department of Computer Science, Technische Universität München, München, Germany
stamatak@cs.tum.edu

Abstract

We are currently faced with the situation where applications have increasing computational demands and there is a wide selection of parallel processor systems. In this paper we focus on exploiting fine-grain parallelism for a demanding Bioinformatics application - MrBayes - and its Phylogenetic Likelihood Functions (PLF) using different architectures. Our experiments compare side-by-side the scalability and performance achieved using general-purpose multi-core processors, the Cell/BE, and Graphics Processor Units (GPU). The results indicate that all processors scale well for larger computation and data sets. Also, GPU and Cell/BE processors achieve the best improvement for the parallel code section. Nevertheless, data transfers and the execution of the serial portion of the code are the reasons for their poor overall performance. The general-purpose multi-core processors prove to be simpler to program and provide the best balance between an efficient parallel and serial execution, resulting in the largest speedup.

1. Introduction

In order to address the constant demand for increased performance within the complexity and power budgets, current microprocessors are composed of multiple cores. Two major challenges need to be addressed in order to efficiently exploit the increasing on-chip parallel resources: the architecture of these large-scale many-core processors and the programmability of such systems.

The multi-core models currently available in the market represent different attempts to address the above issues. The major contribution of our work is to analyze the different existing architectures in terms of performance, scalability and programmability. To achieve this goal we focus on three different types of multi-core architectures and how to exploit fine-grain parallelism for a demanding and relevant application from the area of bioinformatics.

The three different architectures and models under study are: general-purpose homogeneous multi-core (dual- and quad-core Intel and AMD processors); heterogeneous multi-core (IBM Cell/BE); and graphics processors units (NVIDIA GPUs). The general-purpose multi-core and Cell/BE processors support the Multiple-Program Multiple-Data (MPMD) model, while the GPU processors support the Single-Program Multiple-Data (SPMD) model. The memory hierarchy of the different architectures has also interesting distinct characteristics. For the general purpose multi-core processors the caches are completely handled by hardware. In contrast, for the Cell/BE the cache memory of the parallel processing elements is completely handled by software, *i.e.* the programmer has the responsibility of mapping the data and explicitly loading the data before its use. For the GPU processors we have an intermediate solution where the programmer is required to map the data to the cache memory but the accesses are handled by hardware.

Regarding the target application, we assess how MrBayes [10], a program for Bayesian inference of evolutionary (phylogenetic) trees, can benefit from fine-grain parallelism. Phylogenetic inference deals with the reconstruction of the evolutionary history for a set of organisms based on a multiple sequence alignment of molecular sequence data. Due to the large number of potential alternative unrooted binary tree topologies the problem of finding the best scoring tree is NP-hard for the Maximum Likelihood model [2]. The scoring function used in MrBayes is also adopted in

[*] P.Trancoso and L.Sousa are members of HiPEAC (EU FP7 program).

other phylogenetic inference programs [5, 11].

The Phylogenetic Likelihood Function (PLF) in MrBayes is parallelized by using OpenMP for the general purpose multi-core systems, POSIX Threads for the Cell/BE systems, and CUDA for the GPU systems. Experimental results show that all considered architectures scale well their performance when the input data set is increased. In the systems with hardware-managed caches, the sharing of a cache level, within the chip, by all cores, is a determining factor for efficient synchronization and therefore the scalability of the number of calls to the parallel section. The users effort in managing the software-managed caches is compensated by the efficient synchronization mechanisms in the Cell/BE. Moreover, since the PPE was designed only to coordinate the execution of the SPEs, it is not able to exploit single thread performance as the traditional CPUs. As such, its overall performance is affected by the penalty of the serial code execution. Overall, comparing to the serial execution, the GPUs reduce significantly the execution time of the parallel section but the data transfer overheads penalize the effective overall speedup. Therefore, the best overall speedup is achieved by the general-purpose multi-core systems, which combine efficient parallel and serial execution. While our results are based on MrBayes, the work presented here is of general interest, because it discusses programming techniques to efficiently exploit memory-intensive fine-grained loop-level parallelism, and provides a performance comparison across different architectures.

The remainder of this paper is organized as follows. In Section 2 we present the different architectures analyzed in this work. Section 3 describes MrBayes and its parallelization strategies proposed for the different architectures. Section 4 describes the experimental setup and results. Finally, Section 5 covers relevant related work and conclusions are presented in Section 6.

2. Multi-core Architectures

2.1. General-Purpose Multi-Cores

The general-purpose microprocessors support the MPMD model and include different levels of hardware-managed cache memory. While the number of cores is still relatively small, each core is able to exploit the ILP for efficient single thread execution. Regarding the memory hierarchy, the last cache level on the chip is in most cases shared by all the cores. Typically the memory hierarchy is coherent therefore allowing the use of a shared-memory parallel programming model. In addition, the sharing of the internal cache by all cores allows for the efficient data transfer and synchronization between them. For these processors, parallel programming is relatively easy using POSIX Threads (*pthreads*) or OpenMP directives.

2.2. Cell/BE

The Cell/BE is a heterogeneous multi-core architecture consisting of 9 cores: one general purpose core, the PowerPC Processor Element (PPE), and eight special purpose cores, the Synergistic Processing Elements (SPEs). The PPE is a simple processor that was designed for coordinating the execution on the SPEs and run the Operating System (OS). The SPEs are simpler processors as their purpose is the execution of the parallel code. Each SPE includes a small private unified memory, the *Local Store* (LS), with 256KB. A key component of this loosely coupled system is the *Element Interconnect Bus* (EIB). This high-bandwidth, memory-coherent bus allows the cores to communicate through DMA data transfers between local and remote memories. The Cell/BE does not support shared-memory at hardware level, giving the user the responsibility to efficiently manage the memory space. Applications on the Cell/BE can be parallelized using pthreads but, unlike the general-purpose multi-core, the user is also responsible for the necessary data transfers and allocation to the corresponding LS memories.

2.3. Graphics Processing Unit(GPU)

The GPU processors include a large number of very basic cores and are typically used as accelerators to a host system. A GPU is usually connected through a system bus (*e.g.* PCIe) to the CPU, and can be used for general-purpose applications (GPGPU) [7].

The Compute Unified Device Architecture (CUDA) is a compiler-supported programming model that offers an extended version of the C language for programming recent NVIDIA GPUs. Parallelism with CUDA is achieved by executing the same function or kernel by N different CUDA threads which, in turn, are organized in blocks. During execution CUDA threads may access data in multiple levels of the memory hierarchy: private local memory, shared memory and global memory. Data organization in this hierarchical memory is crucial to achieve the best efficiency.

3. Fine-Grain Parallelism in MrBayes

3.1. MrBayes Overview

MrBayes is a popular program for Bayesian phylogenetic inference. This program is based on the Maximum Likelihood (ML) model [3] that represents a broadly accepted criterion to score phylogenetic trees. To compute the Phylogenetic Likelihood Functions (PLF) on a fixed tree one needs to estimate the branch lengths and the parameters of the statistical model of nucleotide substitution. For DNA data sequences, a model of nucleotide substitution is provided by a 4x4 matrix (denoted as Q and shown in Figure 2), that contains the instantaneous transition probabilities for a certain DNA nucleotide (`A` - Adenine, `C` - Cytosine, `G` - Guanine, or `T` - Thymine) to mutate into a nucleotide `A`, `C`, `G`,

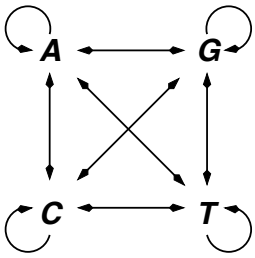


Figure 1: A DNA substitution model

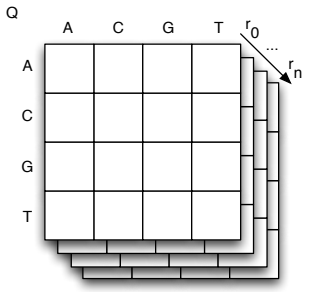


Figure 2: Nucleotide substitution matrix Q

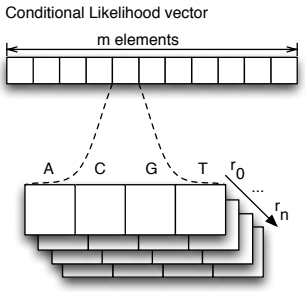


Figure 3: Conditional likelihood (cl) vector (detail of one element)

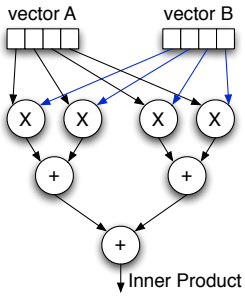


Figure 4: Inner product dependencies graph

or `T`, according to the substitution model in Figure 1. Ideally to compute the nucleotide substitution probabilities for a branch with length t (time along the branch), one has to compute $P(t) = e^{Qt}$.

This basic model is usually extended by additional model parameters, such as the Γ model [14], which typically uses 4 discrete rates, r_0, r_1, r_2, r_3. To compute the likelihood of a fixed *unrooted* tree topology with given branch lengths and model parameters, one initially needs to compute the entries for all internal likelihood vectors. They contain the probabilities of observing an `A`, `C`, `G`, or `T` for each column of the input alignment. Hence, the conditional likelihood vectors "cl" have the same length "m" as the sequences in the input alignment. In the Γ model every likelihood vector element is composed by 16 floating-point numbers: 4 discrete rate arrays each with 4 entries as shown in Figure 3.

Phylogenomic data sets contain a large number of concatenated genes. For example, a current real-world study contains 1,500 genes. The execution of a Maximum Likelihood-based phylogenetic analysis using this real-world example requires approximately 2,000,000 CPU hours on a BlueGene/L system [8]. In this work we exclusively focus on fine-grain loop-level parallelism at the PLF level. All current PLF-based programs spend the largest part of run time, typically around 85-95% in the computation of the PLF [8, 12]. The basic task consists in scheduling and distributing the required likelihood vector data structures and loop iterations to the several processing elements.

The profiling of the execution showed that `CondLikeDown`, `CondLikeRoot`, and `CondLikeScaler` are the three main PLF functions, which account for more than 85% of the total execution time. As depicted in Figure 5, the PLF mainly consist of independent `for` loops with a computational load that depends on the sequence length (m) and the number of discrete rates (r). In each iteration, the functions `CondLikeDown` and `CondLikeRoot` multiply the likelihood vector elements by the substitution matrix for each of the defined discrete rates. Thus, considering 4 discrete rates, the computation of a likelihood element requires 4 matrix-vector multiplications, or 16 inner products. The inner product can be seen as a reduction, namely the multiply and accumulate operations over 2 vectors of 4 floating-point numbers as depicted in Figure 4. Finally, the function `CondLikeScaler` is used to avoid numerical underflow. To scale the entries, this function finds the maximum entry by successive comparisons, which is also a reduction operation and has a computational complexity similar to the previous functions. PLF implementation in MrBayes uses single-precision arithmetic.

Input: cl Arrays
Input: Substitution Matrices Q
Output: Result clP
foreach cl *element* **do**
  foreach *Discrete rate* **do**
    foreach Q *row* **do**
      Calculate Inner Product (Q_{row}, cl)
    end
    Multiply the final arrays.
  end
end

Figure 5: *CondLikeRoot* and *CondLikeDown* computation

3.2. General-Purpose Multi-Core

In most cases, the parallelization of the applications consists in identifying the most costly loops and, provided that the loop iterations are independent, parallelize them via the `#pragma omp parallel for` OpenMP directive. In many cases the code includes nested loops. Based on a profile analysis, it is relatively straight-forward to parallelize the sequential code, since each PLF function contains compute-intensive independent loops. Given the relatively small number of parallel resources utilized in the OpenMP execution, the decision taken was to parallelize the outermost loop, thus reducing the parallelization overheads. Besides, each loop data accesses are analyzed in order to identify shared and private variables in the individual iterations.

3.3. Cell/BE

In the Cell/BE the execution of the PLFs is mapped to the SPEs, while the rest of the application executes on the PPE.

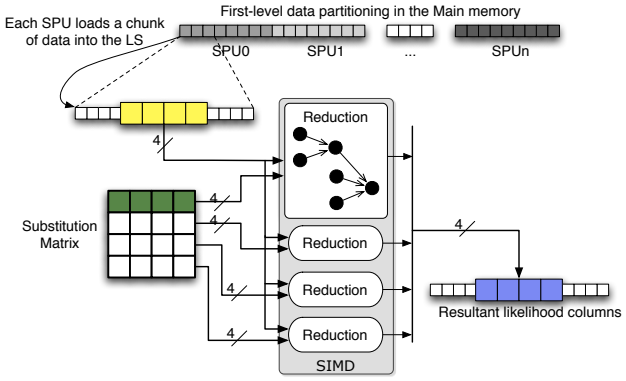


Figure 6: Data partitioning hierarchy and computation of a discrete rate array in the Cell; simplified generic diagram

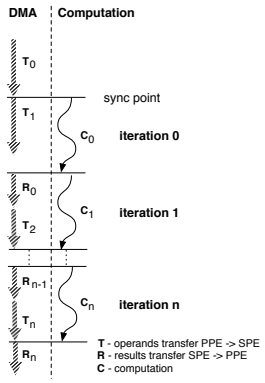


Figure 7: DMA/SPE synchronization

Despite the SPE memory limitations (256KB), the code space required for PLFs is not critical, only 90KB. However, the amount of data to be processed (the likelihood vectors), *per* PLF invocation, does not fit into the local SPE memory. Therefore, a simple data partitioning scheme among the SPEs is not sufficient. To solve this problem, we deploy a two-level partitioning scheme as depicted in Figure 6: (*i*) in a first level partition the PPE evenly assigns the m likelihood vector elements to the different SPEs, the first-level data blocks are then processed in parallel by the *Synergistic Processing Units* (SPUs); (*ii*) according to the blocks' size, each SPE creates smaller sub-partitions (chunks) that are sequentially computed by each SPU. This method allows to achieve full scalability independently of the data size. The PLFs execution on the SPUs is coordinated by a simple local Finite State Machine (FSM) through messages issued by the PPE, namely: to trigger the execution of the PLF functions (see section 3.1), the calculation of the chunk sizes, and to finalize the computation. With a specific message type to calculate the chunk sizes, sequences of data with different sizes can be used at the same time, *i.e.* likelihood vectors with different m's.

Finally, we also use the *Single-Instruction Multiple-Data (SIMD)* hardware structures of the SPEs to exploit the data parallelism in the reductions referred in section 3.1(Figure 4), as presented in Figure 6. Two different approaches can be implemented: *i*) data parallelism is exploited in each of the reductions by directly the computations shown in Figure 4 to the SIMD instructions; and *ii*) the computation is optimized by exploiting data parallelism across the four reductions, i.e., one matrix-vector multiplication. In the former approach there is a row-wise access to the matrix Q, a partial multiplication with the vector cl and finally the accumulation of the partial results; while in the latter approach four serial reductions are computed by the SIMD instructions, requiring a column-wise access to the matrix. SIMD instructions are used more efficiently in the second approach due to the reduction characteristics. We implemented both approaches and observed a benefit of 34% for the total speedup and 2x for the PLF speedup in the latter approach compared to the former. Therefore, herein we only present the results regarding the latter approach.

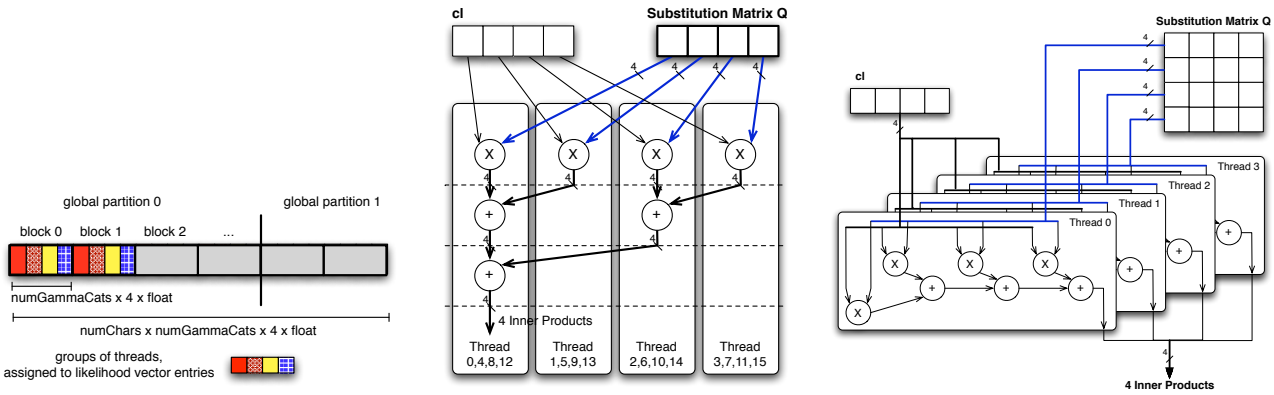
The Cell/BE supports DMA transfers of aligned data for a maximum size of 16KB *per* transfer. To efficiently exploit the data parallelism (SIMD) the likelihood arrays used by each of the three PLF functions are aligned to a 128 byte boundary. For the first mentioned SIMD implementation, dummy elements were inserted to adjust the memory boundaries. For the second approach, besides the dummy elements, the transpose of the substitution matrices Q is calculated to facilitate the columnwise access.

Among the several synchronization mechanisms available on the Cell/BE, *direct problem state accesses* was used for the synchronization between the PPE and the SPEs while DMA transfers were used between the SPEs. The rationale for this choice is that these are the most efficient mechanisms for this particular type of frequent fine-grained synchronization. Direct problem state accesses are similar to mailboxes while with the DMA transfers the PPE performs a busy wait for an SPE notification minimizing the communication/synchronization overhead.

It is important to note that the adopted two-level partitioning method along with the double-buffering technique requires two levels of synchronization to maintain data coherence in the LS and in the main memory. Our communication scheme is outlined in Figure 7: synchronization barriers are depicted by horizontal lines, T_i events represent DMA data transfers to the SPE prior to the corresponding PLF computation C_i, and R_i events represent DMA transfers of the results back to the main memory.

3.4. Graphics Processing Unit (GPU)

For each PLF invocation, the input data is sent to the global GPU memory, the corresponding parallelized PLF function is executed on the GPU side and the results are then transfered back to the CPU. Although the general structure of the algorithm is similar to the one used for the Cell/BE, there are some important differences imposed by the GPU architecture and the API: (*i*) higher amount of parallelism to exploit due to a larger number of processing elements; (*ii*) reduced number of data transfers from the CPU to the de-



(a) GPU data partition example. (b) Thread scheduling, parallelized reduction (c) Thread scheduling, serial reduction

Figure 8: GPU data partition and Likelihood vector computation and synchronization.

vice memory because there is sufficient memory to handle the input data; and (*iii*) data transfer synchronization procedure is handled automatically by CUDA, thus not requiring any additional explicit synchronization mechanism.

To efficiently exploit the GPU architecture, the number of threads must be maximized. As well as for the Cell/BE two different approaches can be followed to distribute the work among the threads: (*i*) directly parallelize each of the reductions described in section 3.1; or (*ii*) parallelize the work at the likelihood vector entry level. Although each of the reduction operations can be performed using a group of threads in parallel as shown in Figure 8(b), this solution requires a large number of synchronization points and conditional statements, which corresponds to a large overhead. In the second approach, the calculation of each vector entry is assigned to one completely independent thread, thus avoiding the overheads of the synchronization points and conditional statements (Figure 8(c)). On the one hand, the amount of concurrency exploited in this latter case is smaller, but on the other hand the threads are completely independent. Also in this case we implemented both approaches and observed a benefit of 36% over the total speedup and 2.5x over the PLF speedup for the latter approach compared to the former one. Therefore, herein we also only present the results relative to the latter.

In order to maximize our design efficiency we had to properly balance the workload by partitioning the data in three levels as depicted in Figure 8(a): (*i*) *global partitions* splits the data when its size is larger than the device memory, to guarantee full scalability; (*ii*) *block partition* evenly distributes the m likelihood array elements between the CUDA blocks, which are processed independently; and (*iii*) *thread partitions* assigns a group of threads to a set of computations. This configuration allows to fully parallelize the likelihood vector computation among the several cores of the GPU in a balanced and scalable way. The same parallelization approach is used in the three PLFs.

The fact that the likelihood vector is organized in multiples of 4 floats can be used to further improve the GPU performance. By assigning groups of 4 threads to each likelihood vector discrete rate (array of 4 floats) allows the compiler to coalesce memory accesses because the threads access to adjacent memory locations. By extending this technique, several groups of 4 threads can access neighbor likelihood vector discrete rates to improve the performance.

Design space exploitation led to testing a wide range of configurations for different number of threads and blocks of the CUDA kernel. The maximum number of threads is directly affected by the maximum shared-memory size, number of registers per thread, and number of threads required to use full thread warps [6]. Based on the execution time for the different configurations, it was concluded that 256 threads and 40 blocks was the best solution to use in the GPU 8800 GT, while for the GPU GTX 285 the best results were obtained with 256 threads and 85 blocks (see Table 1).

4. Experimental Setup and Results

The configuration details of the multi-core systems are provided in Table 1. The architecture denoted as *Baseline* is used as reference for calculating relative figures in the next sub-sections. The OpenMP implementation used in this work was the one supported by the Intel Compiler Suite 11 (C/C++ and Fortran) and executed on the Xeon and Opteron based systems. For the Cell-based systems we used the Cell SDK version 2.0 while for the GPU-based system we used CUDA v2.1 to generate the executables tested in this work.

DNA test data sets of various sizes were generated and simulated with Seq-Gen [9] (v1.3.2), and they were used as inputs for MrBayes version 3.1.2. As input for Seq-Gen we used trees with 10, 20, 50, and 100 leaves obtained from analyses of real data sets and evolved artificial sequence alignments with 500,000 columns each under the GTR+Γ model on those trees [14]. For every tree size, subalignments with 1,000, 5,000, 20,000, and 50,000 distinct columns were automatically extracted by using a perl script. This was done because identical alignment columns can be

Table 1: Systems Setup

	Baseline	**2xXeon(4)**	**4xOpteron(4)**	**8xOpteron(2)**	**PS3**	**QS20**	**8800GT**	**GTX285**
System	Generic	IBM x3650	Dell PowerEdge M905	Sun x4600 M2	Sony PS3	IBM QS20	NVIDIA 8800 GT	NVIDIA GTX 285
Cores	1	2x Quad	4x Quad	8x Dual	1+6	2x (1+8)	112	240
Model	Intel E8400	Intel E5320	AMD 8354	AMD 8218	PPE+SPE	PPE+SPE	Streaming	Streaming
Freq	3.0GHz	1.8GHz	2.2GHz	2.6GHz	3.2GHz	3.2GHz	1.5GHz	1.476GHz
Cache	6MB	2x4MB	4x512KB+2MB	2x1MB	512KB	2x 512KB	256KB	480KB
Mem	2GB	48GB	64GB	64GB	256MB	2x 512MB	512MB	1GB

compressed into column patterns under ML, which are then assigned a respective higher per-pattern weight. Hence, in our experiments the number of columns corresponds exactly to the number of patterns and thus to the length of the compute-intensive `for` loops. In the remaining of the paper we use x_y to denote the number of leaves and columns for each of the input tests. Finally, we also used a subalignment of a real-world phylogenomic alignment of mammalian sequences with 20 organisms, 28,740 alignment columns, and 8,543 distinct column patterns. MrBayes was executed with fixed random number seeds and a fixed number of generations to ensure a fair comparison of the results.

4.1. Scalability

As described previously, each input data set is characterized by two numbers, the *leaves* and the *columns*. Roughly, the *columns* represent the size of the data being computed in the compute-intensive loops while the number of *leaves* is related with the number of calls to the PLF. Thus, we use the increasing number of *leaves* to study the computation intensity scaling and the number of *columns* to study the data size scaling of the algorithm for the different architectures.

4.1.1 General-Purpose Multi-Core Architectures

The first results here presented regard the scalability of the different general-purpose multi-core based systems. In Figure 9 we show the relative speedup for the execution on the 8-core two-way Quad-core Xeon, the 16-core four-way Quad-core Opteron and the 16-core eight-way Dual-core Opteron systems for all input data sets. The relative speedup is the speedup achieved for n-cores *vs* 1-core execution. The chart in Figure 9 includes four groups of data, each representing a different input data size, while each point within a group represents a different amount of computation.

Starting with the two-way Quad-core Xeon (*2xXeon(4)*), the higher speedup values require using larger data sets (lowest speedup of 6 for the 1K *columns* data sets) but this speedup is penalized by the computation intensity, *i.e.* the increasing number of calls to PLF. This indicates that the OpenMP implementation for spawning of the parallel PLF execution and synchronization at the end of the execution, leads to some overhead. To avoid this issue it would be interesting to explore alternative parallel execution using implementations that are more efficient (*e.g.* the TFlux [13] model which has minimal synchronization and runtime overheads).

Regarding the four-way Quad-core Opteron (*4xOpteron(4)*), the results in this chart show that the performance of this particular system suffers for executions with small data sets (1K *columns*). For that case the speedup values are low and unstable. For the larger data sets, the speedup values are larger but the results show again that the increasing computation introduces some penalty on the speedup values. Nevertheless, for this architecture the overheads seem to be smaller, most probably due to the internal architecture of the multi-core chips. While both are Quad-core, in the case of the Xeon, its package contains two dual-core dies while the Opteron includes a single die with four cores. As such, the shared L2 cache in the case of the Xeon is actually composed of two separated caches, each one shared by a couple of cores. In the case of the Opteron the L2 cache is shared by the four cores. The implications of this fact are that the communication between the cores can be more efficient across the four cores of the Opteron.

Finally, for the eight-way Dual-core Opteron (*8xOpteron(2)*) the results show similar trends to the results obtained for the Xeon system. The most relevant factor here, and this becomes more severe with the increasing number of cores, is the penalty suffered for the increasing computation. These results confirm the hypothesis presented in the discussion of the Quad-core Opteron results. Indeed it seems that the spawning and synchronization of the parallel tasks results in a larger overhead as the chips include fewer cores. This is due to the more costly communication between the cores in different chips, and in different dies but on the same package in the Xeon.

Overall, the results for the general-purpose multi-core architectures show that the efficiency of the fine-grain parallelism depends on the integration of more cores into the same die sharing the same cache, therefore allowing for efficient intra-chip, across-core communication.

4.1.2 Cell/BE

The speedup results presented for the two Cell/BE based systems, the *PS3* and the *QS20*, with one and two Cell/BE processors, respectively, are relative to the execution on a single SPE processor. As such, the n-core speedup is the ratio between the execution on 1 SPE and the execution on n SPE processors. Figure 10 depicts the results for both systems. From this chart it is possible to observe that other than for the smallest data set (1K columns), the speedup values are close to the ideal. Actually, the maximum efficiency

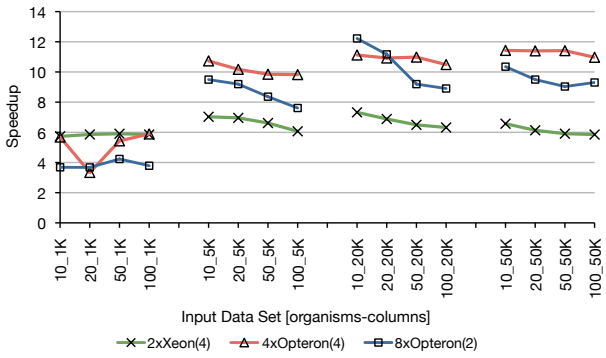


Figure 9: Scalability for the multi-core based system

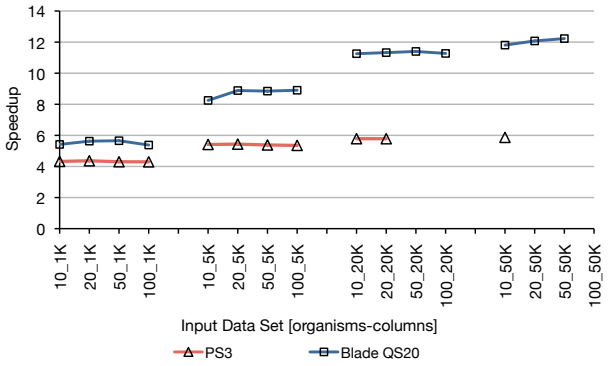


Figure 10: Scalability for the Cell/BE based systems

obtained on the Cell/BE for the PLF is 92% compared to the average 71% obtained for the different general-purpose multi-core systems. Also important to notice is that the performance is stable across the different computation intensities. The results thus show that the Cell/BE is more tolerant to the synchronization overheads mentioned before. It is relevant to mention that unlike the general-purpose multi-core processors, the cores in the Cell/BE do not share any common cache. This though does not result in a penalty for the execution as the Cell/BE has efficient mechanisms to exchange data and perform synchronization. In addition, instead of relying on a automatically generated code, the Cell/BE is executing carefully hand-tuned code. Thus the reduced overheads observed.

Although the speedup value is stable for the 16-core execution, its value is close to 12x. While we expected this value to be higher, it is interesting that the same value is achieved for the best cases with the general-purpose multi-core systems. However, in contrast with the multi-cores, a slight increase in the speedup is observed for the increasing computational intensity. This results from the fact that given the efficient synchronization and data transfers, the higher computation-to-data ratio, the better the performance.

4.1.3 GPU

In the case of the GPU processors, given its model, the execution on a single core is not applicable. Therefore, the speedup reported is the performance improvement relative to the execution on the lower-spec GPU (*8800GT*) using the smaller data set size (*10_1K*). As such, the results here presented should not be directly compared with the results from the previous sections. Instead, these results should be used only to study the trends on how the speedup scales for the changes in data sets and computation intensity.

Figure 11 presents the relative speedup values for the 8800GT and GTX285 based systems, as described before. From this chart it is possible to observe that the speedup values increase as the data sets increase up to the 20K and 50K *columns* where the maximum speedup values are achieved. Interesting is the fact that unlike the general-purpose multi-cores, and at a higher degree of what had been observed for the Cell/BE, for the GPUs, as the computational intensity increases, so does the speedup. This is probably due to the fact that this type of processor is designed for efficient execution of small parallel threads and therefore the overheads observed for the other systems do not apply for the GPUs. It is also well known that GPUs benefit increases when the computation-to-data ratio is high. Finally, it is relevant to notice that there is a significant difference between the speedup achieved by the 8800GT and the GTX285. The speedup achieved by the GTX285 is 2.2x larger for the 20K column sets and up to 2.4x larger for the 50K column sets, in comparison with the speedup achieved for the 8800GT. This results indicates that the GPU architecture scales well as the number of cores available in the GTX285 (240) is 2.1x larger than the number of cores in the 8800GT (112).

4.2. Total Execution Time

In this Section we analyze the total execution time across the different systems. For this analysis we also use the baseline execution as a reference. In addition, in order to have a fairer comparison, we scale the results according to the frequencies of each system and the baseline.

Although all architectures have proven to have good scalability in the execution of the parallel section, the reduction of the total execution time is relatively small, specially for all systems other than the general-purpose multi-core ones. Although not shown, the 8-core system achieves approximately 4x speedup, while the 16-core systems achieve approximately 7x speedup. The Cell/BE systems, both for 6 and 16 cores achieve only about 1.5x speedup and only one of the GPU systems achieves a speedup which is approximately 1.5x.

In order to understand this relatively poor performance we analyzed the execution time for each system, breaking down the frequency-scaled time into the time spent for the execution of the PLF (*plf*) and the remaining application execution time (*Remaining*). Also, for the GPU systems we have identified the portion of the execution time spent in the communication of data to and from the graphics card. The normalized execution time for all systems is presented in Figure 12.

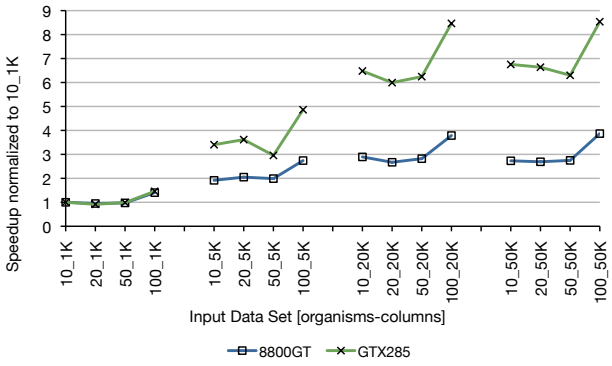


Figure 11: Scalability for the 8800GT and GTX285 systems

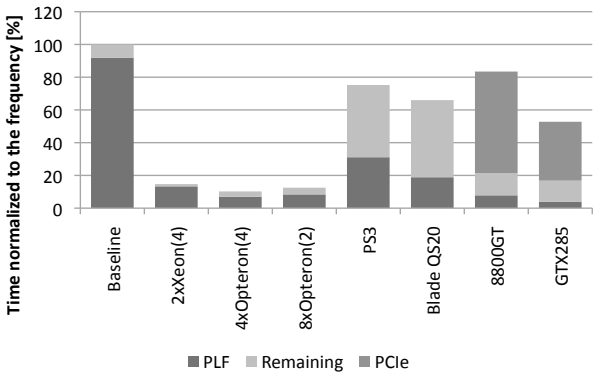


Figure 12: Frequency scaled total time for all systems, real data set

As reported earlier, for the baseline, the larger portion of time (greater than 90%) is consumed in the execution of the PLF. For example, for the real input set the complete execution takes 62s out of which 57s are spent in the execution of the PLF. All parallel architectures target the reduction of this large portion of time. Notice that all architectures are successful in achieving this goal. The general-purpose multi-core systems reduce it to 10-15%, the Cell/BE systems reduce it to 20-30% and the GPU systems to 5-10%. Considering just this goal, the GPUs would be the most successful architecture. Nevertheless, other factors play an important role in the execution of the complete application.

For the GPU systems we observe that the *Remaining* time increases slightly and this is not only due to the host system of the graphics card being slightly slower than the baseline. Before each PLF execution the host needs to coordinate with the card and ship the code for execution on the GPU, which is a relatively small overhead. However, the largest overhead for the GPU systems is the transfer of the data to and from the graphics card. This results in a large penalty in the execution time to such an extent that for the 8800GT its execution time is at the end larger than the baseline. This proves that in order to exploit the performance benefits offered by the GPU architecture it is needed to explore faster ways to transfer the data, or overlap the data transmission with computation on the GPU and host system similarly to what is done for the Cell/BE.

For the Cell/BE, while the PLF execution is also significantly reduced, the *Remaining* time increases significantly compared to the original execution on the baseline system, therefore highly penalizing the total execution time. This is due to the fact that while the PLF code is efficiently executed in parallel on the SPEs, the serial code (*Remainder*) is executed on the PPE, which is a simple core, mainly designed to efficiently coordinate the execution on the SPEs. This analysis is corroborated by the efficiency results, which show a drop of about 40% when comparing the PLF to the execution of the complete application. The differences between the PPE capabilities and the core on the baseline system are, for example, in-order execution and a relatively small 512KB L2 cache. In addition, the PPE is responsible for the coordination and data transfers to the SPEs. It is relevant to notice that although the PPE was designed to be just managing the work executed on the SPEs and executing the host Operating System (OS), its low specifications results in a large penalty even for applications which portion of code to be executed on the PPE is small. To avoid this problem it would be interesting to explore systems with multiple cores in order to use the Cell/BE for the parallel section of the code (as the GPU) and offload the serial execution to more powerful cores.

Finally, the general-purpose multi-cores, although they do not achieve the largest reduction in the PLF execution, they reduce it significantly and at the same time they also reduce the *Remainder* portion of the time. The general-purpose multi-core systems have cores with the similar specifications as the core in the baseline and thus are able to efficiently handle the serial execution. At the same time they have multiple resources therefore allowing to exploit the parallelism required. While it is expected that this type of architecture will not lead to efficient large-scale systems, at the current scale and with the current problem sizes and computation intensity, they are the best solution at hand.

Overall, results observed for the different architectures lead us to believe that a successful future large-scale many-core system will have to be composed of heterogenous cores in order to be able to provide a large degree of parallelism, they should have some sort of shared memory resources or efficient communication mechanisms for synchronization support and also certain powerful cores in order to execute the serial code, even if that is a small portion of the execution. With respect to development times, the Cell/BE is the most demanding system while multi-cores are comparatively easy to program. It took us approximately half a day to parallelize MrBayes on multi-cores, 2 days on the GPU, and 10 days on the Cell/BE.

5. Related Work

With the exception of PBPI [4], that conducts multi-grain Bayesian inference on the BlueGene/L, to the best of our knowledge, no other work has addressed the issue of

parallelizing the PLF. PBPI essentially represents a proof-of-concept work, since the capabilities of the program do not correspond to the needs of Biologists for real-world analyses, mainly, because it only implements the very simple models of nucleotide substitution (see [8] for more details). Most remaining work on parallelizing the PLF has focused on RAxML. The program has recently been parallelized on the BlueGene/L [8], on a variety of multi-core and supercomputer architectures to compare Pthreads, MPI, and OpenMP [12], and on the Cell/BE [1]. The work on the Cell/BE architecture mainly focused on aspects of scheduling multi-grain parallelism and represents proof–of–concept work rather than a production level parallelization of RAxML. The MPI-based parallelization scheme presented on the BlueGene/L [8], is currently being integrated into the publicly available RAxML release. A recent study [12] compared MPI, OpenMP, and Pthreads from multi-core architectures up to an infiniband-connected Linux cluster with SMP-nodes and an SGI Altix 4,700.

In the work presented in this paper, programming paradigms and scalability of the PLF is accessed, from multi-core architectures down to the Cell/BE and GPUs.

6. Conclusions

This work is focused on exploiting emerging multi-core processors and accelerators to improve the performance of the execution of Phylogenetic Likelihood Functions (PLF) from a well-known Bioinformatics application, MrBayes. Given the large number of cores and huge amount of data that will become available in the near future, we decided to focus on exploiting the fine-grain parallelism in the PLFs. While the work focuses on a Bioinformatics application, loop-level parallelism is a common characteristic of many scientific applications. As such, the results found herein are applicable to a wider range of applications.

In our experiments we compare side-by-side the performance achieved using general-purpose multi-core processor, Cell/BE, and Graphics Processor Units (GPU) systems. The experimental results indicate that regarding the execution of the parallel section, all processors scale well. For the general-purpose multi-core, an on-chip shared cache among all cores helps in the scalability by providing fast across core data communication and synchronization. The GPUs are able to achieve the best improvement for the parallel code section but data transfers result in a large penalty. For the Cell/BE, the inefficient execution of the serial portion of the code on the PPE is the reason for the overall poor performance.

While the analysis presented here used only the frequency as the scaling factor, considering other factors such as area and power do not change significantly the conclusions. Overall, the general-purpose multi-core systems achieved the best balance between an efficient parallel and serial execution of the code resulting in the largest speedup for MrBayes. At the same time, this was achieved with the smallest programming effort.

7. Acknowledgements

In order to get the executions on the different systems, in addition to the systems made available from the SiPS and CASPER groups, we would like to thank Bernard Moret for the access to the AMD Barcelona system, the chair of computer architectures and organization TU Munich for the access to the Sun x4600 system, and RZG (Rechenzentrum der Max-Planck gesellschaft) at Munich for the access to the Cell/BE QS20 system.

References

[1] F. Blagojevic, D. Nikolopoulos, A. Stamatakis, and C. Antonopoulos. Dynamic Multigrain Parallelization on the Cell Broadband Engine. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice Of Parallel Programming*, pages 90–100. ACM New York, NY, USA, 2007.

[2] B. Chor and T. Tuller. Maximum Likelihood of Evolutionary Trees: Hardness and Approximation. *Bioinformatics*, 21(1):97–106, 2005.

[3] J. Felsenstein. Evolutionary Trees from DNA Sequences: a Maximum Likelihood Approach. *J. Mol. Evolution*, 17:368–376, 1981.

[4] X. Feng, K. W. Cameron, and D. A. Buell. PBPI: a High Performance Implementation of Bayesian Phylogenetic Inference. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 75, New York, NY, USA, 2006. ACM.

[5] S. Guindon and O. Gascuel. A Simple, Fast, and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood. *Systematic Biology*, 52(5):696–704, 2003.

[6] E. Lindholm, J. Nickolls, et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39–55, March 2008.

[7] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn. GPGPU: General Purpose Computation on Graphics Hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM.

[8] M. Ott, J. Zola, A. Stamatakis, and S. Aluru. Large-scale Maximum Likelihood-based Phylogenetic Analysis on the IBM BlueGene/L. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[9] A. Rambaut and N. Grassly. Seq-Gen: an Application for the Monte Carlo Simulation of DNA Sequence Evolution Along Phylogenetic Trees. *Comp. App. in the Biosciences*, 13(3):235–238, 1997.

[10] F. Ronquist and J. Huelsenbeck. MrBayes 3: Bayesian Phylogenetic Inference under Mixed Models. *Bioinformatics*, 19:1572–1574, 2003.

[11] 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.

[12] A. Stamatakis and M. Ott. Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study. In *Proceedings of the Third IAPR International Conference on Pattern Recognition in Bioinformatics*, pages 424–435. Springer, 2008.

[13] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 25–34, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Z. Yang. Maximum Likelihood Phylogenetic Estimation from DNA Sequences with Variable Rates Over Sites. *Journal of Molecular Evolution*, 39:306–314, 1994.