Modeling Multigrain Parallelism on Heterogeneous Multi-core Processors

Filip Blagojevic, Xizhou Feng,
Kirk W. Cameron and Dimitrios S. Nikolopoulos
Center for High-End Computing Systems
Department of Computer Science
Virginia Tech
{filip,fengx,cameron,dsn}@cs.vt.edu

Abstract

Heterogeneous multi-core processors integrate conventional processing cores with computational accelerators. To maximize performance on these systems, programs must exploit multiple dimensions of parallelism simultaneously, including task-level and data-level parallelism. Unfortunately, parallel program designs with multiple dimensions of parallelism today are ad hoc, resulting in performance that depends heavily on the intuition and skill of the programmer. Formal techniques are needed to optimize parallel program designs. We propose a parallel computational model for steering multi-grain parallelization in heterogeneous multi-core processors. Our model accurately predicts the execution time and scalability of a program using multiple conventional processors and accelerators. The model reveals optimal degrees of multi-dimensional, task-level and data-level concurrency in parallel programs. We use the model to derive mappings of two full computational phylogenetics applications on multi-processors featuring the IBM Cell Broadband Engine.

1 Introduction

To overcome limitations of conventional general-purpose microprocessors, many high-performance systems are capable of offloading computations to special-purpose hardware. These computational accelerators now come in many forms from SIMD co-processors to FPGA boards to chips with multiple specialized cores. We consider a computational accelerator as any programmable device that is capable of speeding up a computation. These devices accelerate important scientific computing operations [1, 2]. Examples of products utilizing computational accelerators are the Cell Broadband Engine from IBM [3], Cray's XD1 [4], the Starbridge Hypercomputer [5], and SGI's FPGA-based NUMA node [6].

Research is currently exploring the integration of accelerators with more conventional parallel programming paradigms and tools [7, 8, 9]. However, the migration path of parallel programming models to accelerator-based architectures is not without obstacles. Accelerators often require platform-specific programming interfaces and formulation of new parallel algorithms to fully exploit the additional hardware. Furthermore, scheduling code on accelerators and orchestrating parallel execution and data transfers between host processors and accelerators is a non-trivial exercise [8].

Consider the problem of identifying the most appropriate programming model and accelerator configuration for a given parallel application. The simplest way to identify the best combination is to exhaustively measure the execution time of all of the possible combinations of programming models and mappings of the application to the hardware. Unfortunately, this technique is not scalable to large, complex systems or large applications. The execution time of a complex application is the function of many parameters. A given parallel application may consist of N phases where each phase is affected differently by accelerators. Each phase can exploit l levels of multi-grain parallelism or any combination thereof such as ILP, TLP, or both. Each phase or level can use any of m different programming models such as message passing, shared memory, and SIMD, or any combination thereof. Accelerator availability or use may consist of c possible configurations. Exhaustive analysis of the execution time for all combinations requires at least $N \times l \times m \times c$ trials.

Computational models allow algorithm design and comparison independent of the underlying hardware. Models of parallel computation have been instrumental in the adoption and use of parallel systems. Unfortunately, commonly used models of parallel computation [10, 11, 12, 13] are not directly applicable to accelerator-based systems. First, the heterogeneous processing common to these systems is not reflected in typical computational models. Second, current models do not capture the effects of multi-grain parallelism. Third, few models account for the effects of using multiple programming models in the same program. Nonetheless, heterogeneity, multi-grain parallelism, and hybrid programming consume both enormous amounts of programming effort, and significant amounts of execution time, if not handled with care. To overcome these deficits, we present a model for multi-grain parallel computation on heterogeneous multi-core processors, built from host and accelerator processing units. We name the model MMGP, for **M**odel of **M**ulti-**G**rain **P**arallelism.

The term multi-grain parallelism refers to application-level parallelism that is nested and encapsulated by procedural and basic block boundaries. For example, an application phase could include parallel tasks where each task contains vectorized and/or parallelized loops. Such an application has multi-grain task and data parallelism at the granularity of procedures and loops. For each layer and source of parallelism in the application, on a given parallel architecture, there is an optimal combination of parallel execution vehicles in the hardware, which best exploits the available parallelism. These execution vehicles may be homogeneous, heterogeneous, physically nested, or disjoint. MMGP decides on how to best map the application parallelism to the entire suite of parallel execution vehicles available in the hardware.

Typically, applications and architecture parallelism are not aligned ideally. For illustration of our techniques and simplicity, we consider architectures with just two levels of parallel execution vehicles. The first level of parallelism corresponds to the computational abilities of Host Processing Units (or HPUs). HPUs are organized as conventional multi-processors, multi-core processors, or a combination thereof, and support coarse-grain parallelism. A single HPU may serve as a gateway to one or more Accelerator Processing Units (or APUs), which comprise the second layer of parallel execution vehicles. APUs support fine-grain data parallelism in tasks assigned to their HPUs. These concepts easily extend to multiple HPUs and additional levels of hardware parallelism. For example, the IBM Cell Broadband Engine [3] contains a front-end multithreaded processor (or HPU) and up to 8 accelerator processors (or APUs). Each APU is capable of SIMD parallelism. A second example, given our definition of accelerators, is a front-end processor (HPU) which offloads work to an FPGA [6] (APU) for accelerated processing of specific numerical functions in hardware. More than one of these hybrid HPU-APU nodes can be interconnected to form a scalable parallel architecture. Figure 1 illustrates an abstract model of the parallel architectures targeted by MMGP.

MMGP is an analytical model which formalizes the process of programming accelerator-based systems and reduces the need for exhaustive measurements. The input to MMGP is an explicitly parallel application, where parallelism is expressed in a machine-independent manner, using common programming libraries and constructs. Upon identification of some key parameters of the application derived from microbenchmarking and profiling of a sequential run, MMGP predicts with reasonable accuracy the execution time with all feasible mappings of the application to

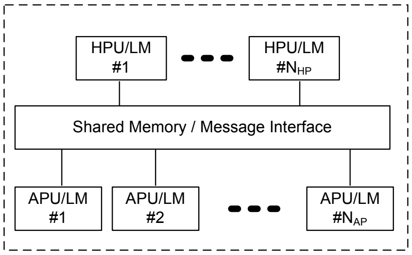


Figure 1: A hardware abstraction of an accelerator-based architecture with two levels of parallelism. Host processing units (HPUs) support coarse grain parallelism. Accelerator processing units (APUs) support finer grain parallelism. Both HPUs and APUs have local memories and communicate through shared-memory or message-passing. Additional levels of parallelism can be expressed hierarchically in similar fashion.

host processors and accelerators. MMGP is fast and reasonably accurate, therefore it can be used to quickly identify optimal operating points, in terms of the exposed layers of parallelism and the degree of parallelism in each layer, on accelerator-based systems. More importantly, MMGP can pin-point the most appropriate programming model to employ for a given parallel application, thereby simplifying substantially the programming effort. Experiments with two complete applications from the field of computational phylogenetics on a shared-memory multiprocessor with Cell BEs, show that MMGP models parallel execution time of complex parallel codes with multiple layers of task and data parallelism, with mean error in the range of 1%–5%, across the feasible program configurations on the target system. In addition to good accuracy in execution time prediction, MMGP identifies with perfect accuracy the best programming model and operating point for each of the codes tested. The codes have contradicting operating points and optimal programming models, thus demonstrating the value of MMGP in speeding up their optimization process.

In the rest of this paper, we establish preliminary background and terminology for introducing MMGP (Section 2), we develop MMGP (Section 3), and we validate MMGP using two computational phylogenetics applications (Section 4). We discuss related work in Section 5 and conclude the paper in Section 6.

2 Modeling Abstractions

Performance can be dramatically affected by the assignment of tasks to resources on a complex parallel architecture with multiple types of parallel execution vehicles. We intend to create a model of performance that captures the important costs of parallel task assignment at multiple levels of granularity, while maintaining simplicity. Additionally, we want our techniques to be independent of both programming models and the underlying hardware. Thus, in this section we identify abstractions necessary to allow us to define a simple, accurate model of parallel computation for accelerator-based architectures.

2.1 Hardware Abstraction

Figure 1 shows our abstraction for accelerator-based architectures. In this abstraction, each node consists of multiple host processing units (HPU) and multiple accelerator processing units (APU). Both the HPUs and APUs have local and shared memory. Multiple HPU-APU nodes form a cluster. We model the communication cost for i and j, where i and j are HPUs, APUs, and/or HPU-APU nodes, using a variant of the *LogP* model [10] of point-to-point communication:

$$C_{i,j} = O_i + L + O_j \tag{1}$$

Where $C_{i,j}$ is the communication cost, O_i and O_j is the overhead of sender and receiver respectively, and L is the communication latency.

In this hardware abstraction, we model an HPU, APU, or HPU-APU node as a sequential device with streaming memory accesses. For simplicity, we assume that additional levels of parallelism in HPUs or APUs, such as ILP and SIMD, can be reflected with a parameter that represents computing capacity. We could alternatively express multi-grain parallelism hierarchically, but this complicates model descriptions without much added value. Assumption of streaming memory accesses, allows inclusion of the effects of communication and computation overlap.

2.2 Application Abstraction

Figure 2 provides an illustrative view of the succeeding discussion. We model the workload of a parallel application using a version of the Hierarchical Task Graph (HTG [14]). An HTG represents multiple levels of concurrency with progressively finer granularity when moving from outermost to innermost layers. We use a phased HTG, in which we partition the application into multiple phases of execution and split each phase into nested sub-phases, each modeled as a single, potentially parallel task. Each subtask may incorporate one or more layers of data or sub-task parallelism. The degree of concurrency may vary between tasks and within tasks.

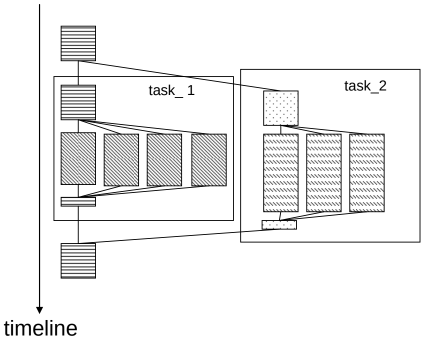


Figure 2: Our application abstraction of two parallel tasks. Two tasks are spawned by the main process. Each task exhibits phased, multi-level parallelism of varying granularity. In this paper, we address the problem of mapping tasks and subtasks to accelerator-based systems.

Mapping a workload with nested parallelism as shown in Figure 2 to an accelerator-based multi-core architecture can be challenging. In the general case, any application task of any granularity could map to any type combination of HPUs and APUs. The solution space under these conditions can be unmanageable.

In this work, we confine the solution space by making some assumptions about the application and hardware. First, we assume that the amount and type of parallelism is known a priori for all phases in the application. In other words, we assume that the application is explicitly parallelized, in a machine-independent fashion. More specifically, we assume that the application exposes all available layers of inherent parallelism to the runtime environment, without however specifying how to map this parallelism to parallel execution vehicles in hardware. In other words, we assume that the application's parallelism is expressed independently of the number and the layout of processors in the architecture. The parallelism of the application is represented by a phased HTG graph. The intent of our work is to improve and formalize programming of accelerator-based multicore architectures. We believe it is not unreasonable to assume those interested in porting code and algorithms to such systems would have detailed knowledge about the inherent parallelism of their application. Furthermore, explicit, processor-independent parallel programming is considered by many as a means to simplify parallel programming models [15].

Second, we prune the number and type of hardware configurations. We assume hardware configurations consist of a hierarchy of nested resources, even though the actual resources may not be physically nested in the architecture. Each resource is assigned to an arbitrary level of parallelism in the application and resources are grouped by level of parallelism in the application. For instance, the Cell Broadband Engine can be considered as 2 HPUs and 8 APUs, where the two HPUs correspond to the PowerPC dual-thread SMT core and APUs to the synergistic (SPE) accelerator cores. HPUs support parallelism of any granularity, however APUs support the same or finer, not coarser, granularity. This assumption is reasonable since it represents faithfully all current accelerator architectures, where front-end processors offload computation and data to accelerators. This assumption simplifies modeling of both communication and computation.

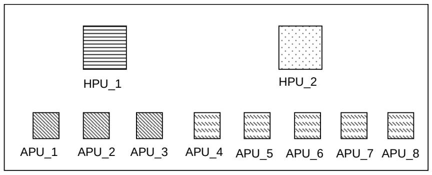


Figure 3: An example mapping of two tasks (see Figure 2) to an accelerator-based system composed of 2 HPUs and 8 APUs. Note the APUs are a shared resource of the HPUs. The assignment problem involves determining how many HPUs and APUs to assign and use for tasks and phases of a parallel application.

3 Model of Multi-grain Parallelism

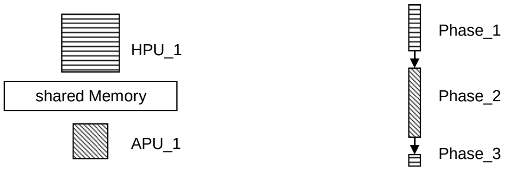
This section provides theoretical rigor to our approach. We present MMGP, a model which predicts execution time on accelerator-based system configurations and applications under the assumptions described in the previous section. Readers familiar with point-to-point models of parallel computation may want to skim this section and continue directly to the results of our execution time prediction techniques discussed in Section 4.

We follow a bottom-up approach. We begin by modeling sequential execution on the HPU, with part of the computation off-loaded to a single APU. Next, we incorporate multiple APUs in the model, followed by multiple HPUs. We end up with a general model of execution time, which is not particularly practical. Hence, we reduce the general model to reflect different uses of HPUs and APUs on real systems. More specifically, we specialize the model to capture the scheduling policy of threads on the HPUs and to estimate execution times under different mappings of multi-grain parallelism across HPUs and APUs. Lastly, we describe the methodology we use to apply MMGP to real systems.

3.1 Modeling sequential execution

As the starting point, we consider the mapping of the program to an accelerator-based architecture that consists of one HPU and one APU, and an application with one phase decomposed into three sub-phases, a prologue and epilogue running on the HPU, and a main accelerated phase running on the APU, as illustrated in Figure 4.

Offloading computation incurs additional communication cost, for loading code and data on the APU, and saving results calculated from the APU. We model each of these communication costs with a latency and an overhead at the end-points, as in Equation 1. We assume that APU's accesses to data during the execution of a procedure are streamed and overlapped with APU computation. This assumption reflects the capability of current streaming architectures, such as the Cell and Merrimac, to aggressively overlap memory latency with computation, using multiple buffers. Due to overlapped memory latency, communication overhead is assumed to be visible only during loading the code



(a) an architecture with one HPU and one APU (b) an application with three phases

Figure 4: The sub-phases of a sequential application are readily mapped to HPUs and APUs. In this example, sub-phases 1 and 3 execute on the HPU and sub-phase 2 executes on the APU. HPUs and APUs are assumed to communicate via shared memory.

and arguments of a procedure on the APU and during returning the result of a procedure from the APU to the HPU. We combine the communication overhead for offloading the code and arguments of a procedure and signaling the execution of that procedure on the APU in one term (O_s), and the overhead for returning the result of a procedure from the APU to the HPU in another term (O_r).

We can model the execution time for the offloaded sequential execution for sub-phase 2 in Figure 4 as:

$$T_{offload}(w_2) = T_{APU}(w_2) + O_r + O_s \tag{2}$$

where $T_{APU}(w_2)$ is the time needed to complete sub-phase 2 without additional overhead. Further, we can write the total execution time of all three sub-phases as:

$$T = T_{HPU}(w_1) + T_{APU}(w_2) + O_r + O_s + T_{HPU}(w_3) \tag{3}$$

To reduce complexity, we replace $T_{HPU}(w_1) + T_{HPU}(w_3)$ with T_{HPU}, $T_{APU}(w_2)$ with T_{APU}, and $O_s + O_r$ with $O_{offload}$. Therefore, we can rewrite Equation 3 as:

$$T = T_{HPU} + T_{APU} + O_{offload} \tag{4}$$

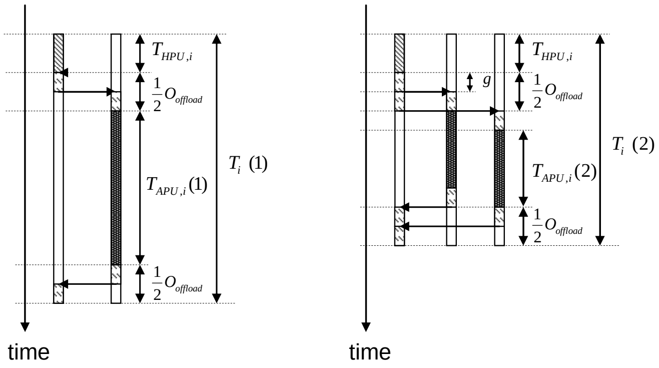
The application model in Figure 4 is representative of one of potentially many phases in an application. We further modify Equation 4 for a generic application with N phases, where each phase i offloads a part of its computation on one APU:

$$T = \sum_{i=1}^{N} \left(T_{HPU,i} + T_{APU,i} + O_{offload} \right) \tag{5}$$

3.2 Modeling parallel execution on APUs

Each offloaded part of a phase may contain fine-grain parallelism, such as task-level parallelism at the sub-procedural level or data-level parallelism in loops. This parallelism can be exploited by using multiple APUs for the offloaded workload. Figure 5 shows the execution time decomposition for execution using one APU and two APUs. We assume that the code off-loaded to an APU during phase i, has a part which can be further parallelized across APUs, and a part executed sequentially on the APU. We denote $T_{APU,i}(1,1)$ as the execution time of the further parallelized part of the APU code during the i^{th} phase. The first index 1 refers to the use of one HPU thread in the execution. We denote $T_{APU,i}(1,p)$ as the execution time of the same part when p APUs are used to execute this part during the i^{th} phase. We denote as $C_{APU,i}$ the non-parallelized part of APU code in phase i. Therefore, we obtain:

$$T_{APU,i}(1,p) = \frac{T_{APU,i}(1,1)}{p} + C_{APU,i} \tag{6}$$



(a) offload to one APU

(b) offload to two APU

Figure 5: Parallel APU execution. The HPU (leftmost bar in parts a and b) offloads computations to one APU (part a) and two APUs (part b). The single point-to-point transfer of part a is modeled as overhead plus computation time on the APU. For multiple transfers, there is additional overhead (g), but also benefits due to parallelization.

Given that the HPU offloads to APUs sequentially, there exists a latency gap between consecutive offloads on APUs. Similarly, there exists a gap between receiving return values from two consecutive offloaded procedures on the HPU. We denote with g the larger of the two gaps. On a system with p APUs, parallel APU execution will incur an additional overhead as large as $p \cdot g$. Thus, we can model the execution time in phase i as:

$$T_i(1,p) = T_{HPU,i} + \frac{T_{APU,i}(1,1)}{p} + C_{APU,i} + O_{offload} + p \cdot g \tag{7}$$

3.3 Modeling parallel execution on HPUs

An accelerator-based architecture can support parallel HPU execution in several ways, by providing a multi-core HPU, an SMT HPU or combinations thereof. As a point of reference, we consider an architecture with one SMT HPU, which is representative of the Cell BE.

Since the compute intensive parts of an application are typically offloaded to APUs, the HPUs are expected to be in idle state for extended intervals. Therefore, multiple threads can be used to reduce idle time on the HPU and provide more sources of work for APUs, so that APUs are better utilized. It is also possible to oversubscribe the HPU with more threads than the number of available hardware contexts, in order to expose more parallelism via offloading on APUs.

Figure 6 illustrates the execution timeline when two threads share the same HPU, and each thread offloads parallelized code on two APUs. We use different shade patterns to represent the workload of different threads.

For m concurrent HPU threads, where each thread uses p APUs for distributing a single APU task, the execution time of a single off-loading phase can be represented as:

$$T_i^k(m,p) = T_{HPU,i}^k(m,p) + T_{APU,i}^k(m,p) + O_{offload} + p \cdot g \tag{8}$$

where $T_i^k(m,p)$ is the completion time of the k^{th} HPU thread during the i^{th} phase.

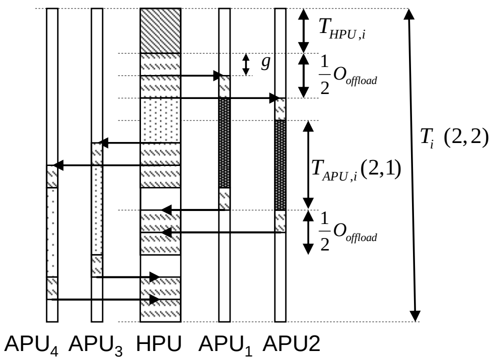


Figure 6: Parallel HPU execution. The HPU (center bar) offloads computations to 4 APUs (2 on the right and 2 on the left). The first thread on the HPU offloads computation to APU1 and APU2 then idles. The second HPU thread is switched in, offloads code to APU3 and APU4, and then idles. APU1 and APU2 complete and return data followed by APU3 and APU4.

3.3.1 Modeling the APU time

Similarly to Equation 6, we can write the APU time of the *k*-th thread in phase *i* in Equation 8 as:

$$T^k_{APU,i}(m,p) = \frac{T_{APU,i}(m,1)}{p} + C_{APU,i} \tag{9}$$

Different parallel implementations may result in different $T_{APU,i}(m,1)$ terms and a different number of offloading phases. For example, the implementation could parallelize each phase among m HPU threads and then offload the work of each HPU thread to p APUs, resulting in the same number of offloading phases and a reduced APU time during each phase, i.e., $T_{APU,i}(m,1) = \frac{T_{APU,i}(1,1)}{m}$. As another example, the HPU threads can be used to execute multiple identical tasks, resulting in a reduced number of offloading phases (i.e., N/m, where N is the number of offloading phases when there is only one HPU thread) and the same APU time in each phase, i.e., $T_{APU,i}(m,1) = T_{APU,i}(1,1)$.

3.3.2 Modeling the HPU time

The execution time of each HPU thread is affected by three factors:

1. Contention between HPU threads for shared resources.

2. Context switch overhead related to resource scheduling.

3. Global synchronization between dependent HPU threads.

Considering all three factors, we can model the execution time of an HPU thread in phase *i* as:

$$T^k_{HPU,i}(m,p) = \alpha_m \cdot T_{HPU,i}(1,p) + l \cdot T_{CSW} + O_{COL} \tag{10}$$

In this equation, the parameter l denotes the number of processes sharing a single execution context on the HPU, T_{CSW} is context switching time on the HPU, and O_{COL} is the time needed for collective communication. We assume that l can be greater than 1, if the program oversubscribes the HPUs, so that it can find more sources of parallelism to offload to APUs. The parameter α_m is introduced to account for contention between threads that share resources on the HPU. On SMT and CMP HPUs, such resources typically include one or more levels of the on-chip cache memory. On SMT HPUs in particular, shared resources include also TLBs, branch predictors and instruction slots in the pipeline.

Contention between threads often introduces artificial load imbalance due to occasional unfair hardware policies of allocating resources between threads.

3.3.3 Synthesis

Combining Equation (8)-(10) and summarizing all phases, we can write the execution time for MMGP as:

$$T(m,p) = \alpha_m \cdot T_{HPU}(1,1) + \frac{T_{APU}(1,1)}{m \cdot p} + C_{APU} + N \cdot (O_{Offload} + l \cdot T_{CSW} + O_{COL} + p \cdot g) \tag{11}$$

Due to limited hardware resources (i.e. number of HPUs and APUs), we further constrain this equation to $m \times p \leq N_{APU}$, where N_{APU} is the number of available APUs. As described later in this paper, we can either measure or approximate all parameters in Equation 11 from microbenchmarks and profiles of sequential runs of the program.

3.4 Using MMGP

Given a parallel application, MMGP can be applied using the following process:

1. Calculate parameters including $O_{Offload}$, α_m, T_{CSW} and O_{COL} using micro-benchmarks for the target platform.

2. Profile a short run of the sequential execution without APU offloading to decide the phases needed to be offloaded and estimate $T_{HPU}(1)$ and $T_{APU}(1,1)$.

3. Solve a special case of Equation 11 (e.g. 7) to find the optimal mapping between application concurrency and HPUs and APUs available on the target platform.

3.5 MMGP Extensions

We note that the concepts and assumptions mentioned in this section do not preclude further specialization of MMGP for higher accuracy. For example, in Section 3.1 we assume computation and data communication overlap. This assumption reflects the fact that streaming processors can very often overlap completely memory access latency with computation. For non-overlapped memory accesses, we can employ a DMA model as a specialization of the overhead factors in MMGP. Also, in Sections 3.2 and 3.3 we assume only two levels of parallelism. MMGP is easily extensible to additional levels but the terms of the equations grow quickly without conceptual additions. Furthermore, MMGP can be easily extended to reflect specific scheduling policies for threads on HPUs and APUs, as well as load imbalance in the distribution of tasks between HPUs and APUs. We claim that our equations and concepts are general. To illustrate the usefulness of our techniques we apply them to a real system. We next present results from applying MMGP to Cell.

4 Experimental Validation and Results

We use MMGP to derive multi-grain parallelization schemes for two bioinformatics applications, RAxML and PBPI, on a shared-memory multiprocessor with two Cell BEs. RAxML and PBPI construct evolutionary trees from DNA or AA sequences, using different optimality criteria for approximating the existentially best trees. RAxML is based on the Maximum Likelihood criterion and uses hill-climbing heuristics and bootstrap analyses to search for the best tree, while PBPI generates tree samples from their posterior probability distribution using Markov chain Monte Carlo methods. Although we are using only two applications in our experimental evaluation, we should point out that

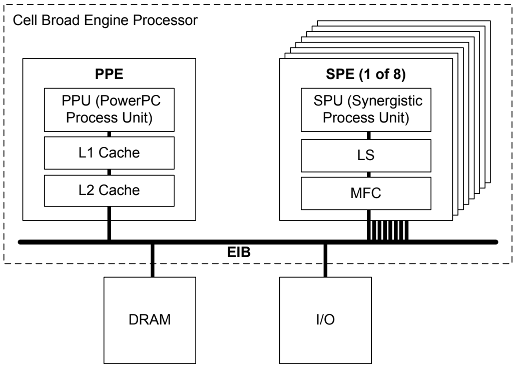

Cell Broad Engine Processor
PPE
PPU (PowerPC Process Unit)
L1 Cache
L2 Cache
SPE (1 of 8)
SPU (Synergistic Process Unit)
LS
MFC
EIB
DRAM
I/O

Figure 7: The IBM Cell Broadband Engine architecture.

these are complete applications used for real-world biological data analyses, and that they are fully optimized for the Cell BE using an arsenal of optimizations, including vectorization, loop unrolling, double buffering, if-conversion and dynamic scheduling [16]. Furthermore, these applications have inherent multi-grain concurrency and non-trivial scaling properties in their phases, therefore scheduling them optimally on Cell is a challenging exercise for MMGP. Lastly, in the absence of comprehensive suites of benchmarks (such as NAS or SPEC HPC) ported on Cell, optimized, and made available to the community by experts, we opted to use codes on which we could verify that enough effort has been invested towards Cell-specific parallelization and optimization. We provide a more detailed discussion of the applications later in this section.

4.1 Experimental Platform

Figure 7 illustrates the organization of Cell, which is the core of our hardware platform. The Cell is a heterogeneous multi-core processor, with accelerator cores used for SIMDization of numerical computation. The processor integrates one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The PPE and SPEs are interconnected with a coherent Element Interconnect Bus (EIB). The PPE is a 64-bit dual-thread SMT processor running the PowerPC ISA, with 32 KB L1 data cache, 32 KB L1 instruction cache, and a 512 KB unified L2 cache. Each SPE is a 128-bit pipelined vector processor with two major components: a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). The SPU has 128 128-bit registers and 256 KB of software-controlled local storage. The SPU implements a Cell-specific SIMD ISA. All single-precision floating point vector instructions on the SPU are fully pipelined, and the SPU can issue one single-precision floating point vector instruction per cycle. Double precision floating point operations are partially pipelined and the processor can execute double precision floating point vector instructions at a maximum throughput of one vector instruction per three cycles. With all eight SPUs active, the Cell is capable of a peak performance of 21 Gflops. With single-precision floating-point arithmetic, the Cell is capable of a peak performance of over 230 Gflops.

The SPEs serve as computation accelerators on Cell and programs are expected to off-load the bulk of their numerical computations on SPEs. An unusual characteristic of SPEs is the software-managed local storage, which enables optimal software-controlled caching, at the expense of increased programming effort. On each SPE, the SPU can fetch instructions and data only from its local storage and write data only to its local storage. The SPEs access main memory through direct memory access (DMA) requests. The DMA transfers are handled by the MFC. Data transferred between local storage and main memory must be 128-bit aligned. The size of each DMA transfer can be at most 16 KB. DMA lists can be used for transferring larger amounts of data (more than 16 KB). A list can have up to 2,048 DMA requests, each for up to 16 KB. The MFC supports only DMA transfer sizes that are 1, 2, 4, 8 or multiples of

16 bytes long. All communications between the PPE, SPE, main memory, and I/O devices are serviced by the EIB, a 4-ring structure that can transmit 96 bytes per cycle and achieves a peak bandwidth of 204.8 Gigabytes/second.

For the experiments presented in this paper, we used a Cell blade with two Cell processors clocked at 3.2 GHz and 512 MB of XDR RAM. The system runs Linux Fedora Core 5 (kernel version 2.6.16), with Cell-specific patches. We compiled our code using GNU Toolchain 4.0.2 and GDB for the Cell SPE/PPE.

4.2 MMGP Parameter approximation

MMGP has six free parameters, C_{APU}, $O_{offload}$, g, T_{CSW}, O_{COL} and α_m. We estimate four of the parameters using micro-benchmarks.

α_m captures contention between processes or threads running on the PPE. This contention depends on the scheduling algorithm on the PPE. We estimate α_m under an event-driven scheduling model which oversubscribes the PPE with more processes than the number of hardware threads supported for simultaneous execution on the PPE, and switches between processes upon each off-loading event on the PPE [8].

To estimate α_m, we use a parallel micro-benchmark that computes the product of two $M \times M$ square matrices consisting of double-precision floating point elements. Matrix-matrix multiplication involves $O(n^3)$ computation and $O(n^2)$ data transfers, thus stressing the impact of sharing execution resources and the L1 and L2 caches between processes on the PPE. We used several different matrix sizes, ranging from 100×100 to 500×500, to exercise different levels of pressure on the thread-shared caches of the PPE. In the MMGP model, we use the mean of α_m obtained from these experiments, which is 1.28.

PPE-SPE communication is optimally implemented through DMAs on Cell. We devised a ping-pong micro-benchmark using DMAs to send a single word from the PPE to one SPE and backwards. We measured PPE$\rightarrow$SPE$\rightarrow$PPE round-trip communication overhead ($O_{offload}$) to 70 ns. To measure the overhead caused by various collective communications we used *mpptest* [17] on the PPE. Using a micro-benchmark that repeatedly executes the *sched_yield()* system call, we estimate the overhead caused by the context switching (T_{CSW}) on the PPE to be 2 μs.

C_{APU} and the gap g between consecutive DMAs on the PPE are application-dependent and cannot be approximated easily with a micro-benchmark. To estimate these parameters, we use a profile of a sequential run of the code, with tasks off-loaded on one SPE.

4.3 Case Study I: Using MMGP to parallelize PBPI

4.3.1 PBPI Implementation

PBPI [18, 19] is a parallel Bayesian phylogenetic inference implementation, which constructs phylogenetic trees from DNA or AA sequences using the Markov chain Monte Carlo sampling method. The method exploits multi-grain parallelism, which is available in Bayesian phylogenetic inference, to achieve scalability on large-scale distributed memory systems, such as the IBM BlueGene/L [20]. The algorithm of PBPI can be summarized as follows:

1. Partition the Markov chains into chains groups; and split the data set into segments along the sequences.

2. Organize the virtual processors that execute the code into a two-dimensional grid; map each chain group to each row on the grid, and map each segment to one column on the grid.

3. During each generation, compute the partial likelihood across all columns, and then use all-to-all communication to collect the complete likelihood values to all virtual processors on the same row.

4. When there are multiple chains, randomly choose two chains for swapping using point-to-point communication.

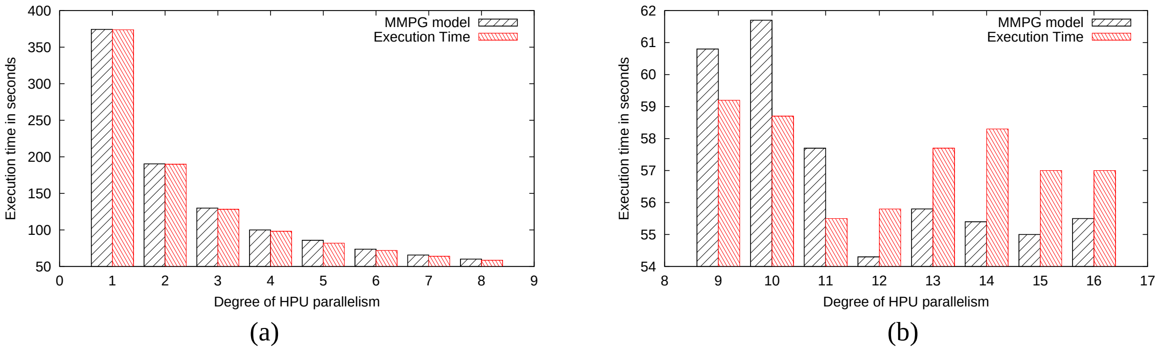


Figure 8: MMGP predictions and actual execution times of PBPI, when the code uses one dimension of PPE (HPU) parallelism, when number of SPEs is between 1 and 8 (a), and number of SPEs is between 9 and 16 (b)

PBPI is implemented in MPI. The approach that we follow to port PBPI to Cell, is to offload the computationally expensive functions (i.e. the likelihood calculation) to the SPE and apply a sequence of Cell-specific optimizations on the off-loaded code, more specifically loop unrolling and vectorization, double buffering for maximum overlap of computation with DMA accesses, and if-conversion to eliminate or parallelize the execution branches wherever possible, to avoid the heavy branch penalties on the SPE. We also developed a custom scheduler for the code running on the PPE.

4.3.2 PBPI with One Dimension of Parallelism

We compare the PBPI execution times predicted by MMGP to the actual execution times obtained on real hardware, using various degrees of PPE and SPE parallelism, i.e. the equivalents of HPU and APU parallelism on Cell. These experiments illustrate the accuracy of MMGP, in a sample of the feasible program configurations. The sample includes one-dimensional decompositions of the program between PPE threads, with simultaneous off-loading of code to one SPE from each PPE thread, one-dimensional decompositions of the program between SPE threads, where the execution of tasks on the PPE is sequential and each task off-loads code which is data-parallel across SPEs, and two-dimensional decompositions of the program, where multiple tasks run on the PPE threads concurrently and each task off-loads code which is data-parallel across SPEs. In all cases, the SPE code is SIMDized in the innermost loops, to exploit the vector units of the SPEs. We believe that this sample of program configurations is representative of what a user would reasonably experiment with while trying to optimize the codes on the Cell.

For these experiments, we used the arch107_L10000 input data set. This data set consists of 107 sequences, each with 10000 characters. We run PBPI with one Markov chain for 20000 generations. Using the time base register on the PPE and the decrementer register on one SPE, we obtained the following model parameters for PBPI: $T_{HPU} = 1.3$s, $T_{APU} = 370$s, $g = 0.8$s and $O = 1.72$s.

Figure 8 compares MMGP and actual execution times for PBPI, when PBPI only exploits one-dimensional PPE (HPU) parallelism in which each PPE thread uses one SPE for off-loading. We execute the code with up to 16 MPI processes, which off-load code to up to 16 SPEs on two Cell BEs. Referring to Equation 11, we set $p = 1$ and vary the value of m between 1 to 8. The X-axis shows the number of processes running on the PPE (i.e. HPU parallelism), and the Y-axis shows the predicted and measured execution times. The maximum prediction error of MMGP is 5%. The arithmetic mean of the error is 2.3% and the standard deviation is 1.4.

Figure 9 illustrates predicted and actual execution times when PBPI uses one dimension of SPE (APU) parallelism. Referring to Equation 11, we set $p = 1$ and vary m from 1 to 8. MMGP remains accurate, the mean prediction error

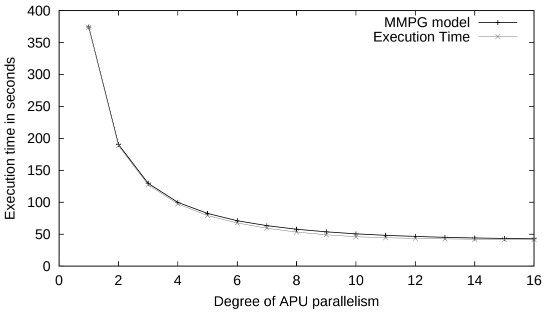


Figure 9: MMGP predictions and actual execution times of PBPI, when the code uses one dimension of SPE (APU) parallelism, with a data-parallel implementation of the maximum likelihood calculation.

is 4.1% and the standard deviation is 3.2. The maximum prediction error in this case is higher (approaching 10%) when the APU parallelism increases and the code uses SPEs on both Cell processors. A closer inspection of this result reveals that the data-parallel implementation of tasks in PBPI stops scaling beyond the 8 SPEs confined in one Cell processor, because of DMA bottlenecks and non-uniformity in the latency of memory accesses by the two Cell processors on the blade. Capturing the DMA bottlenecks requires the introduction of a model of DMA contention in MMGP, while capturing the NUMA bottleneck would require an accurate memory hierarchy model integrated with MMGP. The NUMA bottleneck can be resolved by a better page placement policy implemented in the operating system. We intend to examine these issues in our future work. For the purposes of this paper, it suffices to observe that MMGP is accurate enough despite its generality. As we show later, MMGP predicts accurately the optimal mapping of the program to the Cell multi-processor, regardless of inaccuracies in execution time prediction in certain edge cases.

4.3.3 PBPI with Two Dimensions of Parallelism

Multi-grain parallelization aims at exploiting simultaneously task-level and data-level parallelism in PBPI. We only consider multi-grain parallelization schemes in which $deg\text{HPU} \cdot deg\text{APU} \leq 16$, i.e. the total number of SPEs (APUs) on the dual-processor Cell Blade we used in this study. $deg()$ denotes the degree of a layer of parallelism, which corresponds to the number of SPE or PPE threads used to run the code. Table 1 shows the predicted and actual execution times of PBPI for all feasible combinations of multi-grain parallelism under the aforementioned constraint. Each entry in the table shows actual and predicted execution time. MMGP's mean prediction error is 3.2%, the standard deviation of the error is 2.6 and the maximum prediction error is 10%. The important observation in these results is that MMGP agrees with the experimental outcome in terms of the mix of PPE and SPE parallelism to use in PBPI for maximum performance. In a real program development scenario, MMGP would point the programmer to the direction of using both task-level and data-level parallelism with a balanced allocation of PPE contexts and SPEs between the two layers.

4.4 Case Study II: Using MMGP to Parallelize RAxML

4.4.1 RAxML Implementation

RAxML uses an embarrassingly parallel master-worker programming paradigm, implemented with MPI. In RAxML, workers perform two tasks:

1. calculation of multiple inferences on the initial alignment in order to determine the best known Maximum Likelihood tree;

PPE\SPE	1	2	3	4	5	6	7	8
1	373.8 / 374.4	188.5 / 190.4	127.4 / 129.7	97.4 / 99.9	79.3 / 82.5	67.6 / 71.1	59.3 / 63.4	53.4 / 57.8
2	189.9 / 190.4	98.6 / 99	68.9 / 69.2	54.6 / 54.8	46.6 / 46.5	43.6 / 41.4	40.1 / 38	39.7 / 35.7
3	128.2 / 129.9	67.7 / 69.2	48.4 / 49.7	39 / 40.4	34.3 / 35.3			
4	98.2 / 100	53 / 54.8	38.5 / 40.4	**34.2 / 33.7**				
5	81.9 / 85.9	47.6 / 51	38.3 / 40.7					
6	72 / 73.7	44.5 / 44.9						
7	64 / 65.7	40.8 / 41.4						
8	58.5 / 60.1	40.6 / 39.1						

PPE\SPE	9	10	11	12	13	14	15	16
1	49.1 / 53.7	46 / 50.6	44.4 / 48.2	43.5 / 46.5	42.7 / 45.1	42.1 / 44.1	41.6 / 43.4	41.6 / 42.8

SPE\PPE	9	10	11	12	13	14	15	16
1	59.2 / 60.8	58.7 / 61.7	55.5 / 57.7	55.8 / 54.3	57.7 / 55.8	58.3 / 55.4	57 / 55	57 / 55.5

Table 1: MMGP predictions and actual execution times of PBPI, when the code uses two dimensions of SPE (APU) and PPE (HPU) parallelism. The mix of degrees of parallelism which optimizes performance is shown in bold. The second and third tables illustrate the results when SPE parallelism is scaled to two Cell processors, i.e. the code uses one PPE thread and more than 8 SPE threads (second table), and PPE parallelism is scaled to two Cell processors, i.e. the code uses more than 8 PPE threads, each offloading to one SPE (third table). In each entry, the first figure is actual execution time and the second figure is predicted execution time.

2. bootstrap analyses to determine how well supported are some parts of the Maximum Likelihood tree.

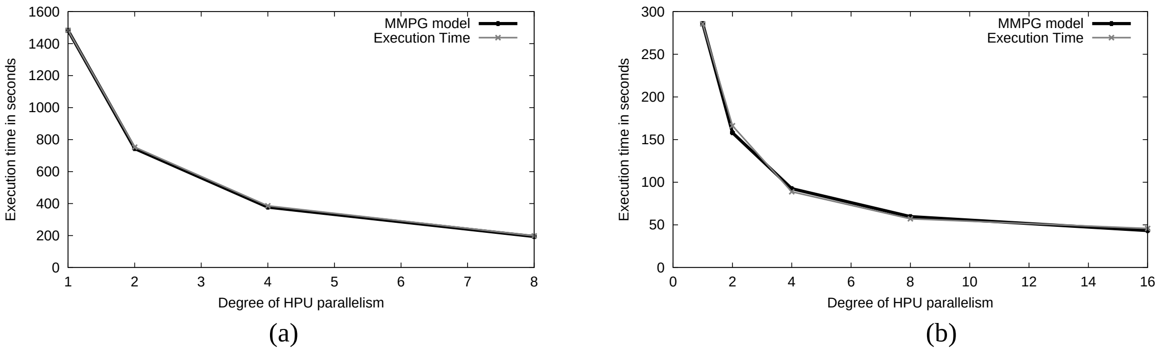
From a computational point of view, inferences and bootstraps are identical. We use an optimized port of RAxML on Cell, described in further detail in [16].

4.4.2 RAxML with a Single Layer of Parallelism

The units of work (bootstraps) in RAxML are distributed evenly between MPI processes, therefore the degree of PPE (HPU) concurrency is bound by the number of MPI processes. As discussed in Section 3.3, the degree of HPU concurrency may exceed the number of HPUs, so that on an architecture with more APUs than HPUs, the program can expose more concurrency to APUs. The degree of SPE (APU) concurrency may vary per MPI process. In practice, the degree of PPE concurrency can not meaningfully exceed the total number of SPEs available on the system, since as many MPI processes can utilize all available SPEs via simultaneous off-loading. Similarly to PBPI, each MPI process in RAxML can exploit multiple SPEs via data-level parallel execution of off-loaded tasks across SPEs. To enable maximal PPE and SPE concurrency in RAxML, we use a version of the code scheduled by a Cell BE event-driven scheduler [8], in which context switches on the PPE are forced upon task off-loading and PPE processes are served with a fair-share scheduler, so as to have even chances for off-loading on SPEs.

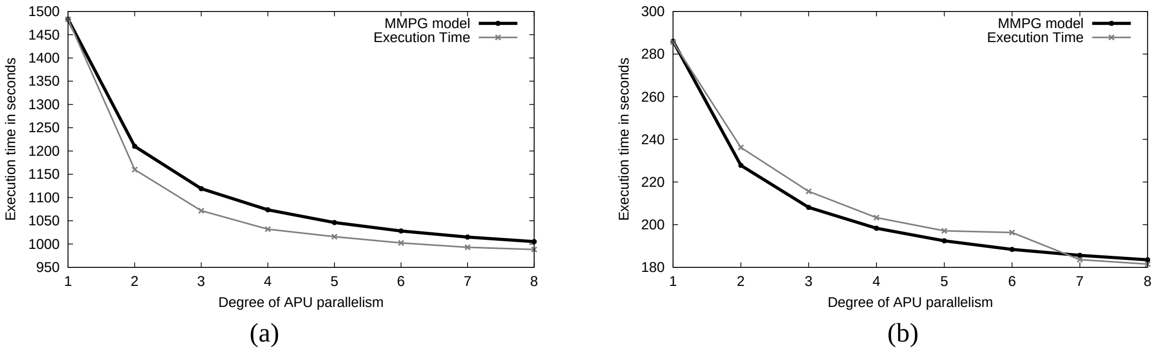
We evaluate the performance of RAxML when each process performs the same amount of work, i.e. the number of distributed bootstraps is divisible by the number of processes. The case of unbalanced distribution of bootstraps between MPI processes can be handled with a minor modification to Equation 11, to scale the MMGP parameters by a factor of $\frac{(\lceil\frac{B}{M}\rceil\cdot M)}{B}$, where B is the number of bootstraps (tasks) and M is the number of MPI processes used to execute the code.

We compare the execution time of RAxML to the time predicted by MMGP, using two input data sets. The first data set contains 10 organisms, each represented by a DNA sequence of 20,000 nucleotides. We refer to this data set as DS1. The second data set (DS2) contains 10 organisms, each represented by a DNA sequence of 50,000 nucleotides. For both data sets, we set RAxML to perform a total of 16 bootstraps using different parallel configurations.



(a) (b)

Figure 10: MMGP predictions and actual execution times of RAxML, when the code uses one dimension of PPE (HPU) parallelism: (a) with DS1, (b) with DS2.



(a) (b)

Figure 11: MMGP predictions and actual execution times of RAxML, when the code uses one dimension of SPE (APU) parallelism: (a) with DS1, (b) with DS2.

The MMGP parameters for RAxML, obtained from profiling a sequential run of the code are $T_{HPU} = 3.3$s, $T_{APU} = 63$s, $C_{APU} = 104$s for DS1, and $T_{HPU} = 8.8$s, $T_{APU} = 118$s, $C_{APU} = 157$s for DS2. The values of other MMGP parameters are negligible compared to T_{APU}, T_{HPU}, and C_{APU}, therefore we disregard them for RAxML. Note that the off-loaded code that cannot be parallelized (C_{APU}) takes 57-62% of the execution time of a task on the SPE. Figure 10 illustrates the estimated and actual execution times of RAxML with up to 16 bootstraps, using one dimension of PPE (HPU) parallelism. In this case, each MPI process offloads tasks to one SPE and SPEs are utilized by oversubscribing the PPE with more processes than the number of hardware threads available on the PPEs. For DS1, the mean MMGP prediction error is 7.1%, the standard deviation is 6.4. and the maximum error is 18%. For DS2, the mean MMGP prediction error is 3.4%, the standard deviation is 1.9 and the maximum error is 5%.

Figure 11 illustrates estimated and actual execution times of RAxML, when the code uses one dimension of SPE (APU) parallelism, with a data-parallel implementation of the maximum likelihood calculation functions across SPEs. We should point out that although both RAxML and PBPI perform maximum likelihood calculations in their computational cores, RAxML's loops have loop-carried dependencies that prevent scalability and parallelization in many cases [16], whereas PBPI's core computation loops are fully parallel and coarse enough to achieve scalability. The limited scalability of data-level parallelization of RAxML is the reason why we confine the executions with data-level parallelism on at most 8 SPEs. As shown in Figure 11, the data-level parallel implementation of RAxML does not scale substantially beyond 4 SPEs. When only APU parallelism is extracted from RAxML, for DS1 the mean MMGP prediction error is 0.9%, the standard deviation is 0.8. and the maximum error is 2%. For DS2, the mean MMGP prediction error is 2%, the standard deviation is 1.3 and the maximum error is 4%.

4.4.3 RAxML with Two Dimensions of Parallelism

Table 2 shows the actual and predicted execution times in RAxML, when the code exposes two dimensions of parallelism to the system. Once again, regardless of execution time prediction accuracy, MMGP is able to pin-point the optimal parallelization model, which in the case of RAxML is task-level parallelization with no further data-parallel decompositions of tasks between SPEs, as the opportunity for scalable data-level parallelization in the code is limited. Innermost loops in tasks are still SIMDized within each SPE. MMGP remains accurate, with mean execution time prediction error of 4.3%, standard deviation of 4, and maximum prediction error of 18% for DS1, and mean execution time prediction error of 2.8%, standard deviation of 1.9, and maximum prediction error of 7% for DS2. It is worth noting that although the two codes tested are fundamentally similar in their computational core, their optimal parallelization model is radically different. MMGP accurately reflects this disparity, using a small number of parameters and rapid prediction of execution times across a large number of feasible program configurations.

PPE╲SPE	**1**	2	3	4	5	6	7	8
1	172 / 170.3	139.2 / 138.8	128.3 / 128.3	123.4 / 123	121.5 / 119.9	119.4 / 117.8	119.4 / 116.3	118.5 / 115.1
2	97.4 / 90.1	78.7 / 74.36	73 / 69.1	69.8 / 66.4	68.2 / 64.9	67.8 / 63.8	67.5 / 63.1	67.8 / 62.5
4	53.2 / 49.5	44.6 / 41.6	41.7 / 39	40.7 / 37.7				
8	28 / 29.2	23.6 / 25.3						
16	**16.1** / **19.1**							

PPE╲SPE	**1**	2	3	4	5	6	7	8
1	285.6 / 286	236.2 / 227.8	215.6 / 208.1	203.3 / 198.3	197.1 / 192.4	196.3 / 188.4	183.6 / 185.6	181.5 / 183.5
2	166.2 / 158.1	127.2 / 128.6	115.2 / 118.7	114.7 / 113.8	113.8 / 110.9	112 / 108.9	109.1 / 107.5	108.6 / 106.5
4	88.8 / 92.5	73 / 77.7	67.5 / 72.8	65.9 / 70.4				
8	57.1 / 59.7	51.9 / 52.4						
16	**45.8** / **43.3**							

Table 2: MMGP predictions and actual execution times of RAxML, when the code uses two dimensions of SPE (APU) and PPE (HPU) parallelism. The mix of degrees of parallelism which optimizes performance is shown in bold. In each entry, the first figure is actual execution time and the second figure is predicted execution time. The table on the top illustrates predicted and actual execution times for DS1, while the table at the bottom illustrates predicted and actual execution times in DS2.

5 Related Work

In this section we review related work in programming environments and models for parallel computation on conventional homogeneous parallel systems and programming support for nested parallelization. The list of related work in models for parallel computation is by no means complete, but we believe it provides adequate context for the model presented in this paper.

Traditional parallel programming models, such as BSP [11], LogP [10], PRAM [12] and derived models [21, 22, 23, 24] developed to respond to changes in the relative impact of architectural components on the performance of parallel systems, are based on a minimal set of parameters to capture the impact of communication overhead on computation running across a homogeneous collection of interconnected processors. MMGP borrows elements from LogP and its derivatives, to estimate performance of parallel computations on heterogeneous parallel systems with multiple dimensions of parallelism implemented in hardware. A variation of LogP, HLogP [13], considers heterogeneous clusters with variability in the computational power and interconnection network latencies and bandwidths between the nodes. Although HLogP is applicable to heterogeneous multi-core architectures, it does not consider nested parallelism. It should be noted that although MMGP has been evaluated on architectures with heterogeneous processors, it can readily support architectures with heterogeneity in their communication substrates as well (e.g. architectures providing both shared-memory and message-passing communication).

Several parallel programming models have been developed to support nested parallelism, including nested parallel languages such as NESL [25], task-level parallelism extensions to data-parallel languages such as HPF [26], extensions of common parallel programming libraries such as MPI and OpenMP to support nested parallel constructs [27, 28], and techniques for combining constructs from parallel programming libraries, typically MPI and OpenMP, to better exploit nested parallelism [29, 30, 31]. Prior work on languages and libraries for nested parallelism based on MPI and OpenMP is largely based on empirical observations on the relative speed of data communication via cache-coherent shared memory, versus communication with message passing through switching networks. Our work attempts to formalize these observations into a model which seeks optimal work allocation between layers of parallelism in the application and optimal mapping of these layers to heterogeneous parallel execution hardware. NESL [25] and Cilk [32] are languages based on formal algorithmic models of performance that guarantee tight bounds on estimating performance of multithreaded computations and enable nested parallelization. Both NESL and Cilk assume homogeneous machines.

Subhlok and Vondran [33] present a model for estimating the optimal number of homogeneous processors to assign to each parallel task in a chain of tasks that form a pipeline. MMGP has a similar goal of assigning co-processors to simultaneously active tasks originating from the host processors, however it also searches for the optimal number of tasks to activate in host processors, in order to achieve a balance between supply from host processors and demand from co-processors. Sharapov et. al [34] use a combination of queuing theory and cycle-accurate simulation of processors and interconnection networks, to predict the performance of hybrid parallel codes written in MPI/OpenMP on ccNUMA architectures. MMGP uses a simpler model, designed to estimate scalability along more than one dimensions of parallelism on heterogeneous parallel architectures.

Research on optimizing compilers for novel microprocessors, such tiled and streaming processors, has contributed methods for multi-grain parallelization of scientific and media computations. Gordon et. al [35] present a compilation framework for exploiting three layers of parallelism (data, task and pipelined) on streaming microprocessors running DSP applications. The framework uses a combination of fusion and fission transformations on data-parallel computations, to ”right-size” the degree of task and data parallelism in a program running on a homogeneous multi-core microprocessor. Eichenberger et. al [36] and Zhao and Kennedy [37] present several compiler optimizations that expose nested parallelism on the Cell Broadband Engine. Blagojevic et. al [8] present runtime scheduling algorithms for exploiting nested parallelism, also on the Cell. MMGP is a complementary tool which can assist both compile-time and runtime optimization on heterogeneous multi-core platforms. The development of MMGP coincides with several related efforts on measuring, modeling and optimizing performance on the Cell Broadband Engine [38, 39]. An analytical model of the Cell presented by Williams et. al [40], considers execution of floating point code and DMA accesses on the Cell SPE for scientific kernels parallelized at one level across SPEs and vectorized further within SPEs. MMGP models the use of both the PPE and SPEs and has been demonstrated to work effectively with complete application codes. In particular, MMGP factors the effects of PPE thread scheduling, PPE-SPE communication and SPE-SPE communication into the Cell performance model.

6 Conclusions

The introduction of accelerator-based parallel architectures complicates the problem of mapping algorithms to systems, since parallelism can no longer be considered as a one-dimensional abstraction of processors and memory. We presented a new model of multi-dimensional parallel computation, MMGP, which we introduced to relieve users from the arduous task of mapping parallelism to accelerator-based architectures. We have demonstrated that the model is fairly accurate, albeit simple, and that it is extensible and easy to specialize for a given architecture. We envision three uses of MMGP: i) As a rapid prototyping tool for porting algorithms to accelerator-based architectures. More specifically, MMGP can help users derive not only a decomposition strategy, but also an actual mix of programming models to use in the application in order to best utilize the architecture, while using architecture-independent programming techniques. ii) As a compiler tool for assisting compilers in deriving efficient mappings of programs to accelerator-based architectures automatically. iii) As a runtime tool for dynamic control of parallelism in applications, whereby the runtime system searches for optimal program configurations in the neighborhood of optimal configurations derived by MMGP, using execution time sampling or prediction-based techniques. Extensions of MMGP which we will explore in future research include accurate modeling of non-overlapped communication and memory accesses, accurate modeling of SIMD and instruction-level parallelism within accelerators, integration of the model with runtime performance prediction and optimization techniques, and application of the model to both conventional and unconventional accelerator-based parallel systems, including systems comprising FPGAs.

Acknowledgments

This research is supported by the National Science Foundation (Grant CCR-0346867), the U.S. Department of Energy (Grants DE-FG02-05ER25689, DE-FG02-06ER25751), and equipment funds from the College of Engineering at Virginia Tech.

References

[1] S. Craven and P. Athanas. Examining the Viability of FPGA Supercomputing. *EURASIP Journal on Embedded Systems*, 2007.

[2] J. Tripp, A. Hanson, M. Gokhale, and H. Mortveit. Partitioning Hardware and Software for Reconfigurable Supercomputing Applications. In *Proc. of Supercomputing'2005*, Seattle, WA, November 2005.

[3] IBM Corporation. Cell Broadband Engine Architecture, Version 1.01. Technical report, October 2006.

[4] M. Fahey, S. Alam, T. Dunigan, J. Vetter, and P. Worley. Early Evaluation of the Cray XD1. In *Proc. of the 2005 Cray Users Group Meeting*, 2005.

[5] Starbridge Systems. A Reconfigurable Computing Model for Biological Research: Application of Smith-Waterman Analysis to Bacterial Genomes. Technical report, 2005.

[6] R. Chamberlain, S. Miller, J. White, and D. Gall. Highly-Scalable Recondigurable Computing. In *Proc. of the 2005 MAPLD International Conference*, Washington, DC, September 2005.

[7] P. Bellens, J. Perez, R. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proc. of Supercomputing'2006*, Tampa, FL, November 2006.

[8] F. Blagojevic, D. Nikolopoulos, A. Stamatakis, and C. Antonopoulos. Dynamic Multigrain Parallelization on the Cell Broadband Engine. In *Proc. of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–100, San Jose, CA, March 2007.

[9] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proc. of Supercomputing'2006*, Tampa, FL, November 2006.

[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Scauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 1–12, San Diego, California, May 1993.

[11] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 22(8):103–111, August 1990.

[12] P. Gibbons. A More Practical PRAM Model. In *Proc. of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, Santa Fe, NM, June 1989.

[13] J. Bosque and L. Pastor. A Parallel Computational Model for Heterogeneous Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 17(12):1390–1400, December 2006.

[14] M. Girkar and C. Polychronopoulos. The Hierarchical Task Graph as a Universal Intermediate Representation. *International Journal of Parallel Programming*, 22(5):519–551, October 1994.

[15] K. Asanovic, R. Bodik, C. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California–Berkeley, December 2006.

[16] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. Raxml-cell: Parallel phylogenetic tree inference on the cell broadband engine. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, Long Beach, CA, March 2007.

[17] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Proc. of the 6th European PVM/MPI User's Group Meeting*, pages 11–18, Barcelona, Spain, September 1999.

[18] X. Feng, K. Cameron, and D. Buell. PBPI: a high performance Implementation of Bayesian Phylogenetic Inference. In *Proc. of Supercomputing'2006*, Tampa, FL, November 2006.

[19] X. Feng, D. Buell, J. Rose, and P. Waddell. Parallel algorithms for bayesian phylogenetic inference. *Journal of Parallel Distributed Computing*, 63(7-8):707–718, 2003.

[20] X. Feng, K. Cameron, B. Smith, and C. Sosa. Building the Tree of Life on Terascale Systems. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, Long Beach, CA, March 2007.

[21] K. Cameron and X. Sun. Quantifying Locality Effect in Data Access Delay: Memory LogP. In *Proc. of the 17th International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[22] A. Alexandrov, M. Ionescu, C. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model: One Step Closer towards a Realistic Model for Parallel Computation. In *Proc. of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, June 1995.

[23] C. Moritz and M. Frank. LoGPC: Modeling Network Contention in Message Passing Programs. In *Proc. of the 1998 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 254–263, Madison, WI, June 1998.

[24] F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *Proc. of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 133–142, Snowbird, UT, June 2001.

[25] G. Blelloch, S. Chatterjee, J. Harwick, J. Sipelstein, and M. Zagha. Implementation of a Portable Nested Data Parallel Language. In *Proc. of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 102–112, San Diego, CA, June 1993.

[26] J. Subhlok and B. Yang. A New Model for Integrated Nested Task and Data Parallelism. In *Proc. of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, Las Vegas, NV, June 1997.

[27] F. Cappello and D. Etiemble. MPI vs. MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, November 2000.

[28] G. Krawezik. Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. In *Proc. of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 118–127, San Diego, CA, June 2003.

[29] E. Ayguadé, X. Martorell, J. Labarta, M. González, and N. Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. In *Proc. of the 1999 International Conference on Parallel Processing (ICPP'99)*, pages 172–180, Aizu, Japan, August 1999.

[30] A. Gerndt, S. Sarholz, M. Wolter, D. An Mey, C. Bischof, and T. Kuhlen. Particles and Continuum – Nested OpenMP for Efficient Computation of 3D Critical Points in Multiblock Data Sets. In *Proc. of Supercomputing'2006*, Tampa, FL, November 2006.

[31] T. Rauber and G. Ruenger. Library Support for Hierarchical Multiprocessor Tasks. In *Proc. of Supercomputing'2002,* Baltimore, MD, November 2002.

[32] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an Efficient Multithreaded Runtime System. In *Proc. of the 5th ACM Symposium on Principles and Practices of Parallel Programming (PPoPP'95)*, pages 207–216, Santa Barbara, California, August 1995.

[33] J. Subhlok and G. Vondran. Optimal Use of Mixed Task and Data Parallelism for Pipelined Computations. *Journal of Parallel and Distributed Computing*, 60(3):297–319, March 2000.

[34] I. Sharapov, R. Kroeger, G. Delamater, R. Cheveresan, and M. Ramsay. A Case Study in Top-Down Performance Estimation for a Large-Scale Parallel Application. In *Proc. of the 11th ACM SIGPLAN Symposium on Pronciples and Practice of Parallel Programming*, pages 81–89, New York, NY, March 2006.

[35] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data and Pipelined Parallelism in Stream Programs. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, San Jose, CA, October 2006.

[36] A. Eichenberger, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, and B. So. Optimizing Compiler for the CELL Processor. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Saint Louis, MO, September 2005.

[37] Y. Zhao and K. Kennedy. Dependence-based Code Generation for a Cell Processor. In *Proc. of the 19th International Workshop on Languages and Compilers for Parallel Computing*, New Orleans, LA, November 2006.

[38] F. Petrini, G. Fossum, J. Fernandez, A. Varbanescu, M. Kistler, and M. Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, Long Beach, CA, March 2007.

[39] T. Chen, Z. Sura, K. O'Brien, and K. O'Brien. Optimizing the Use of Static Buffers for DMA on a Cell Chip. In *Proc. of the 19th International Workshop on Languages and Compilers for Parallel Computing*, New Orleans, LA, November 2006.

[40] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The Potential of the Cell Processor for Scientific Computing. In *Proc. of the 3rd Conference on Computing Frontiers*, pages 9–20, Ischia, Italy, June 2006.