A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies

Scott Schneider Jae-Seung Yeom Benjamin Rose John C. Linford
Adrian Sandu Dimitrios S. Nikolopoulos *

Department of Computer Science
Virginia Tech
Blacksburg, VA 24060, USA
{scschnei, jyeom, bar234, jlinford, asandu, dsn}@cs.vt.edu

Abstract

On multiprocessors with explicitly managed memory hierarchies (EMM), software has the responsibility of moving data in and out of fast local memories. This task can be complex and error-prone even for expert programmers. Before we can allow compilers to handle the complexity for us, we must identify the abstractions that are general enough to allow us to write applications with reasonable effort, yet specific enough to exploit the vast on-chip memory bandwidth of EMM multi-processors. To this end, we compare two programming models against hand-tuned codes on the STI Cell, paying attention to programmability and performance. The first programming model, Sequoia, abstracts the memory hierarchy as private address spaces, each corresponding to a parallel task. The second, Cellgen, is a new framework which provides OpenMP-like semantics and the abstraction of a shared address spaces divided into private and shared data. We compare three applications programmed using these models against their hand-optimized counterparts in terms of abstractions, programming complexity, and performance.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages

General Terms Design, Languages

Keywords Cell BE, Explicitly Managed Memory Hierarchies, Programming Models

1. Introduction

Multi-core processors with explicitly managed memory hierarchies have recently emerged as general-purpose high-performance computing platforms. Some of these processors, such as Cell and GPUs [5, 21], originated in the computing domains of games and graphics. More recently, processor vendors for mainstream computing markets such as Intel and AMD introduced similar designs [24]. All of these processors have data-parallel components as accelerators. This acceleration is achieved through several to many scalar or SIMD cores, high on-chip bandwidth, and explicit data transfers between fast local memories and external DRAM. Explicit data transfers enable the programmer to use optimal caching policies and multiple streaming data buffers that allow overlap computation with communication [6].

Managing the memory hierarchy in multi-core processors introduces trade-offs in terms of performance, code complexity, and optimization effort. Multi-core processors based on coherent hardware-managed caches provide the abstraction of a single shared address space, which is convenient for parallel programming. However, hardware-managed caches only allow the programmer to improve locality implicitly, through reordering of instructions or realignment of data in main memory. In contrast to hardware managed caches, software-managed local memories introduce per-core, disjoint address spaces that the software is responsible for keeping coherent. Because it is the software's responsibility to manage data, the programmer can explicitly manage locality. The programmer can decide what and when data is placed in local memories, what data gets replaced and what the data layout is in local memories, which can differ from the layout of data in off-chip DRAM [1].

In this work, we analyze and evaluate support for expressing parallelism and locality in programming models for multiprocessors with explicitly managed memory hierarchies. We use the Cell processor as an experimental testbed, which we describe in more detail in Section 2. We study three programming models motivated by and developed for the Cell processor. We explore abstractions for specifying parallel tasks, specifying the working sets of parallel tasks, controlling task granularity, and scheduling data transfers to and from local memories. In particular, we compare programming models where managing parallelism and locality is implicit versus programming models where managing parallelism and locality is explicit. The exploration space of this paper is summarized in Table 1. The programming models are described in more detail along with examples in Section 3. We use three realistic and non-trivial parallel applications for our evaluation: PBPI [11], Fixedgrid [17, 23, 14] and CellStream [22], a variation of the stream benchmark to measure maximum on-chip data transfer bandwidth on the Cell. These applications are described further in Section 4; they stress both computational power and on-chip/off-chip memory bandwidth.

Our experimental analysis, presented in Section 5, shows that programming models where management of locality and paral-

* Also with the Institute of Computer Science, Foundation of Research and Technology – Hellas (FORTH-ICS).

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.

Programming model	Task creation	Granularity	Locality	Scheduling data transfers
Sequoia [10]	explicit	explicit	explicit	implicit
Cellgen	implicit	implicit	implicit	implicit
Cell SDK 3.0 [15]	explicit	explicit	explicit	explicit

Table 1: Programming model exploration space.

lelism is implicit are not necessarily inferior to programming models where the management of locality and parallelism is explicit. Implicit programming models can benefit from a compiler that can analyze references to data residing in off-chip memory and schedule data transfers so as to optimize data-computation overlap. We present a framework providing these capabilities in the context of the Cellgen programming model.

We stipulate that programming models with implicit management of parallelism and locality help programmer productivity, since the programmer writes less code; is concerned with a high-level view of locality (i.e. what data is shared and what data is private, instead of who produces and consumes the data); and relies more on the compiler and the runtime environment for managing parallelism and locality. For parallel applications dominated by data parallelism, we find that explicit task to core mapping is not always necessary. Furthermore, explicit partitioning of the data set of programs into working sets, an arguably tedious process, can be automated in the compiler/runtime environment for codes with affine accesses to arrays.

High-level programming models introduce inevitable overhead in the runtime system. We find that in some cases, this overhead is attributed to incompleteness or lack of functionality. Functionalities lacking in the programming models we explore include shared data among tasks in a computation context, dynamic mapping of tasks to cores for load balancing, defining multiple tasks with different data requirements, and participation of the master thread in computation. In other cases, overheads are imposed by the abstraction provided by the programming model, such as the transformation of simple array accesses to data transfers and local storage accesses. Minimizing these overheads is essential for developing unintrusive and scalable programming models.

2. Hardware Environment

The Cell is a heterogeneous multi-core processor [5]. One of the cores, the Power Processing Element (PPE), is a 64-bit two-way SMT PowerPC core with 32 KB of L1-I and L2-D cache, and 512-KB of L2 cache. The other eight cores on the Cell are homogeneous, 128-bit SIMD-RISC processors. They are called Synergistic Processing Elements (SPEs) and they are typically used as accelerators of data-intensive computation. Each processor has a 128-bit data path, 128 128-bit registers, and 256 KB of software-managed local store. The SPEs can issue two instructions per cycle into two (odd/even) pipelines. One pipeline implements floating point instructions, whereas the other implements branches, load/stores and channel communication instructions, which we will describe shortly.

The SPEs execute code and load/store data exclusively out of their local stores. They communicate with off-chip memory and with the local stores of other SPEs through Direct Memory Accesses (DMAs). DMAs are posted with channel commands from the SPE core to the Memory Flow Controller (MFC). Each MFC has a queue which can hold up to 16 outstanding DMA requests, each of which can send or receive op to 16 KB of contiguous data. The effective addresses of DMAs are translated to physical addresses using the PowerPC memory translation mechanisms.

The PPE, SPEs, memory controller and I/O controller are interconnected via the Element Interconnect Bus (EIB). The EIB has two rings moving data clockwise, two rings moving data counter-clockwise, and one address ring. The EIB operates at half the processor clock frequency and its maximum theoretical bandwidth is 204.8 GB/s. (This assumes all units connected to the EIB can snoop one address per EIB cycle and for each snooped address the EIB transfers a maximum of 128 bytes.) Actual point-to-point bandwidth ranges from 78 to 197 GB/s, depending on several factors, including the position of the units (SPEs, PPE, controllers) on the ring.

Off-chip memory is based on XDR Rambus technology and off-chip memory access bandwidth is 25.6 GB/s inbound/outbound.

```
#pragma cell unroll(16) SPE_start(0) SPE_stop(N/4)        \
        reduction(+: double l = lnL)                      \
        private(double* freq = model->daStateFreqs)       \
        shared(double* sroot = tree->root->siteLike,      \
               int* weight = g_ds.compressedWeight.v)
{
  int i;
  for (i = SPE_start; i < SPE_stop; i++) {
    double temp;
    temp = sroot[(i*4)]*freq[0] + sroot[(i*4)+1]*freq[1] +
           sroot[(i*4)+2]*freq[2] + sroot[(i*4)+3]*freq[3];
    temp = log(temp);
    l +=  weight[i] * temp;
  }
}
```

Figure 1: Cellgen code example of the likelihood calculation in PBPI (see Section 4.3).

3. Programming Models

In this section, we discuss the characteristics and specific implementations of the two programming models in this study. We also contrast these methods with programming for the Cell directly using the Software Development Toolkit.

3.1 Cellgen

Cellgen implements a subset of OpenMP on the Cell [18]. An example of Cellgen code taken from PBPI [11] is given in Figure 1. The model uses a source-to-source optimizing compiler. Programmers identify parallel sections of their code in the form of loops accessing particular segments of memory. Programmers need to annotate these sections to mark them for parallel execution, and to indicate how the data accessed in these sections should be handled. This model provides the abstraction of a shared-memory architecture and an indirect and implicit abstraction of data locality, via the annotation of the data set accessed by each parallel section. While the data sets used by each parallel section are annotated, the code inside these regions is not; it is written in the same way it would be for a sequential program.

Data is annotated as private or shared, using the same keywords as in OpenMP. Private variables follow OpenMP semantics. They are copied into local stores using DMAs and each SPE gets a private copy of the variable. Shared variables are further classified internally in the Cellgen compiler as *in*, *out*, or *inout* variables, using reference analysis. This classification departs from OpenMP semantics and serves as the main vehicle for managing locality on Cell. *In* data need to be streamed into the SPE's local store, *out* data

needs to be streamed out of local stores, and *inout* data needs to be streamed both in and out of local stores. By annotating the data referenced in the parallel section, programmers implicitly tell Cellgen what data they want transferred to and from the local stores. The Cellgen compiler takes care of managing locality, by triggering and dynamically scheduling the associated data transfers. Note also that the abstraction of shared memory is not implemented on top of a coherent software cache [2]. Coherence and locality are managed transparently by the compiler/runtime system, per user directives annotating data accesses.

Being able to stream *in*/*out*/*inout* data simultaneously in Cellgen is paramount for two reasons: the local stores are small, so they can only contain a fraction of the working sets of parallel sections; and the DMA time required to move data in and out of local stores may dominate performance. Overlapping DMAs with computation is necessary to achieve high performance. Data classified by the compiler as *in* or *out* are streamed using double-buffering, while *inout* data are streamed using triple buffering. The number of states a variable can be in determines the depth of buffering. *In* variables can be either streaming in, or computing; *out* variables can be either computing or streaming out; *inout* variables can be streaming in, computing, or streaming out. The Cellgen compiler creates a buffer for each of these states. For array references inside parallel sections, the goal is to maximize computation/DMA overlap by having different array elements in two (for *in* and *out* arrays) or three (for *inout* arrays) states simultaneously.

Overlapping the computation and communication hides the memory transfer latency. This latency can be further hidden by unrolling loops, and Cellgen unrolls loops according to a programmer provided unroll factor. One of our goals in these experiments is to determine if we can deduce an unroll factor implicitly. In conventional loop unrolling, the limiting factor is the number of available registers; register spillage causes hits to the cache and main memory. On a Cell SPE, the limiting factor are the DMAs; the time it takes to transfer data to main memory dominates any effects from using too many registers. By unrolling loop iterations, more data can be transferred in a single DMA, amortizing the startup costs of the transfer.

SPEs operate on independent loop iterations in parallel. Cellgen, like OpenMP, assumes that it is the programmer's responsibility to ensure that loop iterations are in fact independent. However, scheduling of loop iterations to SPEs is done by the compiler. The current implementation uses static scheduling, where the iterations are divided equally among all SPEs. We anticipate introducing dynamic loop scheduling policies in the near future.

3.2 Sequoia

The second class of programming models that we consider in this study expresses parallelism through explicit task and data subdivision. We use Sequoia [10] as a representative of these models. In Sequoia, the programmer constructs trees of dependent tasks where the inner tasks call tasks further down the tree; the real computation typically occurs in leaf tasks. At each level, the data is decomposed and copied to the child tasks as specified, which enforces the model that each task has a private address space. Figure 2 repeats the example of Figure 1 using Sequoia.

Locality is strictly enforced by Sequoia because tasks can only reference local data. In this manner, there can be a direct mapping of tasks to the Cell architecture where the SPE local storage is divorced from the typical memory hierarchy. By providing a programming model where tasks operate on local data, and providing abstractions to subdivide data and pass it on to subtasks, Sequoia is able to abstract away the underlying architecture from programmers. Sequoia allows programmers to explicitly define data and computation subdivision through a specialized notation. Using

```
void task<leaf> Sum::Leaf(in double A[L], inout double B[L])
{
  B[0] += A[0];
}

void task<inner> Likelihood::Inner(in double sroot[N],
 in double freq[M],in int weight[P], out double lnL[L])
{
  tunable T;
  mapreduce(unsigned int i = 0 : (N+T-1)/T) {
    Likelihood(sroot[i*T;T], freq[0;3],
               weight[i*T/4;T/4], reducearg<lnL, Sum>);
  }
}

void task<leaf> Likelihood::Leaf(in double sroot[N],
 in double freq[M], in int weight[P], inout double lnL[L])
{
  unsigned int i,j;
  double temp;

  for(i = 0; i < P; i++) {
    j = 4 * i;
    temp = sroot[j] * freq[0] + sroot[j+1] * freq[1] +
           sroot[j+2] * freq[2] + sroot[j+3] * freq[3];
    temp = log(temp);
    lnL[0] += weight[i] * temp;
  }
}
```

Figure 2: Sequoia code example of the likelihood calculation in PBPI (see Section 4.3).

these definitions, the Sequoia compiler generates code which divide and transfer the data between tasks and performs the computations on the data as described by programmers for the specific architecture. The mappings of data to tasks and tasks to hardware are fixed at compile time.

In Sequoia, application users may suggest to the compiler certain optimization approaches such as double buffering for transferring data and an alternative strategy of mapping data divisions to subtasks. The compiler generates optimized code based on these hints, if possible.

While one of the goals of Sequoia is to free the programmer from having to be aware of underlying data transfer mechanisms, the current Sequoia runtime system does not support transferring non-contiguous data. This is a shortcoming of Cellgen as well. Sequoia tries to ensure that programmers are free from the awareness of the architectural constraints such as the DMA size and data alignment requirements by providing an interface to allocate arrays. Programmers are expected to use such an interface to allocate arrays which are handled by Sequoia. When a programmer makes a request for an array of a particular size, the amount actually allocated by Sequoia may be larger. It must be at least 16 bytes to satisfy the DMA size constraint, and it must be a multiple of 16 bytes to satisfy the DMA alignment constraint. Additionally, Sequoia must allocate memory for the data structure that describes the array.

Sequoia also provides an interface to copy an ordinary array to an array allocated by the Sequoia interface. When applying the Sequoia framework to a given reference code, programmers allocate Sequoia arrays and copy existing arrays to the Sequoia counterparts before calling the Sequoia computation kernel. Conforming to this assures that the architectural constrains are satisfied at the cost of additional copying overhead.

In our experiments, we use the customized array allocation interface which allocates space for the Sequoia array structure only—it does not actually allocate any extra data. We then manually point the Sequoia array data structure to the existing data. Therefore, the DMA constraints are explicitly taken care of in our experiments when an array is allocated. This technique avoids unneces-

	serial	Cell SDK	Sequoia	Cellgen
CellStream	195	+602	+78	+9
Fixedgrid	5,565	+1,040	+214	+34
PBPI	8,623	+721	+165	+5

Table 2: Lines of code for each application using each programming model. In case of Sequoia, we counted the lines in mapping files as well as those in Sequoia source files. We do not count comments or empty lines.

sary copies from application managed data to Sequoia managed data, as well as unnecessary allocations.

3.3 Cell SDK

Our third programming model is the Cell SDK 3.0, as provided by IBM. The SDK exposes architectural details of the Cell to the programmer, such as SIMD intrinsics for SPE code. It also provides libraries for low-level, Pthread style thread-based parallelization, and sets of DMA commands based on a get/put interface for managing locality and data transfers.

Programming in the Cell SDK is analogous, if not harder, than programming with MPI or POSIX threads on a typical cluster or multiprocessor. The programmer needs both a deep understanding of thread-level parallelization and a deep understanding of the Cell hardware.

While programming models can transparently manage data transfers, the Cell SDK requires all data transfers to be explicitly identified and scheduled by the programmer. Furthermore, the programmer is solely responsible for data alignment, for setting up and sizing buffers to achieve computation/communication overlap, and for synchronizing threads running on different cores. However, hand-tuned parallelization also has well-known advantages. A programmer with insight into the parallel algorithm and the Cell architecture can maximize locality, eliminate unnecessary data transfers and schedule data and computation on cores in an optimal manner.

4. Applications

For our analysis, we use three applications: a memory bandwidth benchmark and two realistic supercomputer-class applications. We present these applications in the order of decreasing size of the granularity of parallelism that they exploit on Cell.

4.1 CellStream

We developed a memory bandwidth benchmark called CellStream to understand how to maximize SPE to main memory data transfers. It was designed so that a small computational kernel can be dropped in to perform work on data as it streams through SPEs. If no computational kernel is used and synchronization is ignored, the benchmark is able to match the performance of the sequential read/write DMA benchmark bundled with the Cell SDK 3.0 [16].

We implemented a reference version which runs solely on the PPE. It shares the same design as those that use an SPE, except that all work is performed on the PPE. The purpose of this implementation is to provide a baseline for comparison.

4.1.1 Parallelization with Cell SDK

Data is processed through a three stage pipeline, where a separate PPE thread is used for each stage. The first stage uses a PPE thread to read a file from disk into buffers of a predetermined size. The second stage offloads each buffer to the SPEs to process, and the third stage writes the processed data to an output file.

The benchmark can use the SPEs in various ways to measure the bandwidth of different parts of the architecture. Each SPE reads in data from a source, process that data and then passes it on to a destination. The source and destination can be main memory or another SPE. Data streams can be created by specifying where each SPE should expect input data from and send output data to. Using this framework, the bandwidth inside of the Cell BE can be utilized efficiently.

The PPE has three buffers that it rotates among the three threads so each has a buffer to work on. The SPE uses double buffering with fencing to continuously stream data in and out of the SPE. Fencing DMA calls are able to fill a local SPE buffer as soon as its previous contents are committed back to main memory without requiring the SPE to wait. CellStream uses DMA transfer sizes of 16 KB to maximize transfer bandwidth.

4.1.2 Parallelization with Cellgen

The Cellgen version of CellStream shares the same three-stage pipeline design as the hand-written version. The difference is in how the data is transferred to the SPE. The minimal processing kernel is placed inside a Cellgen code section, and the directive explicitly sets the DMA buffer size to 16 KB. The code generated by Cellgen is similar to that of the hand-written version, so the size, number and order of the DMA calls are the same.

4.1.3 Parallelization with Sequoia

Like the Cellgen version, the Sequoia version of CellStream only concerns itself with streaming data through a single SPE while leaving the behavior of the PPE the same. A leaf task is defined to set every byte in the target array to a certain value in order to achieve this behavior.

4.2 Fixedgrid

Fixedgrid is a comprehensive prototypical atmospheric model written entirely in C. It describes chemical transport via a third order upwind-biased advection discretization and second order diffusion discretization [17, 23, 14]. An implicit Rosenbrock method is used to integrate a 79-species SAPRC'99 atmospheric chemical mechanism for VOCs and NOx on every grid point [4]. Chemical or transport processes can be selectively disabled to observe their effect on monitored concentrations.

To calculate mass flux on a two-dimensional domain, a two-component wind vector, horizontal diffusion tensor, and concentrations for every species of interest must be calculated. To promote data contiguity, Fixedgrid stores the data according to function. The latitudinal wind field, longitudinal wind field, and horizontal diffusion tensor, are each stored in a separate $N_X \times N_Y$ array, where N_X and N_Y are the width and height of the domain, respectively. Concentration data is stored in a $N_S \times N_X \times N_Y$ array, where N_S is the number of monitored chemical species. To calculate ozone (O_3) concentrations on a 600×600 domain as in our experiments, approximately 1,080,000 double-precision values (8.24 MB) are calculated at each time step and 25,920,000 double precision values (24.7 MB) are used in the calculation.

4.2.1 Parallelization

To apply the discretized transport equations at a given point, we only require information at that point and at that point's immediate neighbors in the dimension of interest. This allows us to apply the discretization equations to each dimension independently, and is known as *dimension splitting*. Fixedgrid uses dimension splitting to reduce the two-dimensional problem into a set of one-dimensional problems, and to introduce a high degree of parallelism. Using this method, the discretization equations can be implemented as a single computational routine and applied to each row and column of the concentration matrix individually and in parallel.

Dimension splitting introduces a local truncation error. Fixedgrid employs first-order time splitting to reduce the truncation error. A half time step is used when calculating mass flux along the

domain's x-axis, and a whole time step is used when calculating mass flux along the domain's y-axis. This process is equal to calculating the mass flux in one step, but reduces truncation error by $\mathbf{O}(h^2)$ [14]. Note that this doubles the work required to calculate mass flux for a full row of the concentration matrix. Consequently, we want row discretization to be efficient. All atmospheric data is ordered to keep rows contiguous in memory, thus exploiting locality and facilitating DMA transfers between main memory and SPE local storage.

The block nature of Fixedgrid's domain makes it ideally parallelizable across many CPUs. Distributing a large matrix computation across many CPUs is a thoroughly explored problem, so we consider execution on only one Cell node.

4.2.2 Parallelization with Cell SDK

Fixedgrid has been parallelized and hand-tuned for the Cell Broadband Engine using the function offloading model. The discretized transport equations are offloaded to the SPEs, and PPE-SPE communication is done through DMAs. For each time step, the PPE divides the domain into as many blocks as there are SPEs and passes each SPE a pointer to each block. The PPE instructs the SPE to perform row-order or column-order discretization, and the SPE loops over as many rows/columns are in the block, calculating transport on each row/column. The PPE idles until the SPE signals that the entire block has been processed.

The Cell's stringent memory alignment and contiguity requirements make column discretization much more complex than row discretization. Row data is contiguous and 128-byte aligned, and therefore can be fetched directly from main memory with simple DMA calls. Column data is non-contiguous and, due to the 8-byte size of a double precision float, approximately half the columns will always be incorrectly aligned in memory.

There are two solutions for transferring column data to and from main memory: the PPE may buffer and reorder the data before the SPE fetches it, or the SPE may use DMA lists to fetch the data directly.

Reordering the data on the PPE is much simpler programmatically, and it recovers wasted cycles. However, this bottlenecks the flow of data to the SPEs, since a single process is responsible for reordering data for many concurrently-executing processes.

Alternatively, the SPEs may use DMA lists to fetch and reorder the data directly. This greatly increases the complexity of the program for two reasons. First, DMA lists must be generated for each new column before the data is fetched. Second, the data is interleaved across columns when it arrives in local storage, so vector intrinsics must be used to manipulate the interleaved data. However, the benefit is that distributing the data reordering process facilitates a higher data flow.

Memory transfers are triple-buffered so each SPE simultaneously fetches the next row/column of data, calculates transport on data in local storage, and writes previously-calculated results back to main memory.

4.2.3 Parallelization with Cellgen

The original approach taken for porting Fixedgrid to Cell maps cleanly to the OpenMP style of data parallelism that Cellgen provides. The discretized transport equations were already implemented as a data parallel computation, and the original data decomposition among SPEs is the same as provided by Cellgen. The only difficulty is that the current implementation of Cellgen does not support column accesses. In order to overcome this, we reorder the column data into contiguous memory before and after the Cellgen regions.

Once the data has been re-ordered, the Cellgen directive only needs to indicate the range of the iteration variables and specify which variables are shared and which are private.

4.2.4 Parallelization with Sequoia

Sequoia has the same difficulty in dealing with column access. Although it has language constructs for accessing non-contiguous locations of an array, the current runtime implementation of Sequoia does not support it. Therefore, column data is reordered before and after the Sequoia entry point task is called. Each of the column discretization functions is defined as a task. The tunable data array division is set to the length of the column or row, as required by a single discretization kernel call.

4.3 PBPI

PBPI is a parallel implementation of the Bayesian phylogenetic inference method, which constructs phylogenetic trees from DNA or AA sequences using a Markov chain Monte Carlo (MCMC) sampling method. The computation time of a Bayesian phylogenetic inference based on MCMC is determined by two factors: the length of the Markov chains for approximating the posterior probability of the phylogenetic trees and the computation time needed for evaluating the likelihood values at each generation. The length of the Markov chains can be reduced by developing improved MCMC strategies to propose high quality candidate states and to make better acceptance/rejection decisions; the computation time per generation can be sped up by optimizing the likelihood evaluation and exploiting parallelism. PBPI implements both techniques, and achieves linear speedup with the number of processors for large problem sizes.

For our experiments, we used a data set of 107 taxa with 19,989 nucleotides for a tree. There are three computational loops that are called for a total of 324,071 times and account for the majority of the execution time of the program. The first loop accounts for 88% of the calls, and requires 1.2 MB to compute a result of 0.6 MB; the second loop accounts for 6% of the calls and requires 1.8 MB to compute a result of 0.6 MB; and the third also accounts for 6% of the calls and requires 0.6 MB to compute a result of 8 bytes.

4.3.1 Parallelization

PBPI uses Metropolis-coupled MCMC (MC^3) to explore multiple areas of potentially optimal phylogenetic trees while maintaining a fast execution time. There are two natural approaches to exploiting parallelism in Metropolis-coupled MCMC: chain-level parallelization and sub-sequence-level parallelization. Chain-level parallelization divides chains among processors; each processor is responsible for one or more chains. Subsequence-level parallelization divides the whole sequence among processors; each processor is responsible for a segment of the sequence, and communications contribute to computing the global likelihood by collecting local likelihood from all processors. PBPI combines these two-approaches and maps the computation task of one cycle into a two-dimensional grid topology.

The processor pool is arranged as an $c \times r$ two-dimensional Cartesian grid. The data set is split into c segments, and each column is assigned one segment. The chains are divided into r groups, and each row is assigned one group of chains. When $c = 1$, the arrangement becomes chain-level parallel; when $r = 1$, the arrangement becomes subsequence-level parallel.

PBPI was originally implemented in MPI, and was subsequently ported to Cell. The Cell implementation exploits data parallelism present in subsequence-level computations by off-loading the computationally expensive likelihood estimation functions to SPEs and applying Cell-specific optimizations on the off-loaded code such as double buffering and vectorization.

	PPE only	Cell SDK	Sequoia	Cellgen
GBytes/sec	1.24	6.49	6.46	6.44

Table 3: Data bandwidth through the PPE in the PPE only implementation and through a SPE in three parallel implementations of CellStream.

In all of our experiments, PBPI is configured to run with a single chain and a single group such that parallelization is achieved only by data decomposition on multiple SPEs.

4.3.2 Parallelization with Cellgen

PBPI was parallelized with Cellgen by first identifying the most computationally expensive loops in the program. The loops involved in the maximum likelihood calculation account for more than 98% of the total execution of PBPI. These loops are called frequently, and operate on a large enough data set that there is benefit from streaming the data. We can also determine from inspection that the loop iterations are independent.

Each of the three loops is annotated with a Cellgen pragma which specifies which variables are private to the loop, which variables are shared among the iterations, the stopping and starting conditions of the loop, and the reduction variables, if any. We experimentally determined the best unroll factor for each loop, and specified that unroll factor in the Cellgen directive.

4.3.3 Parallelization with Sequoia

The same loops as the ones chosen by Cellgen for parallelization are defined as tasks. The tunable value for data division determines the DMA buffer size for data arrays. We experiment with various tunable settings to see how they affect performance.

5. Performance Analysis

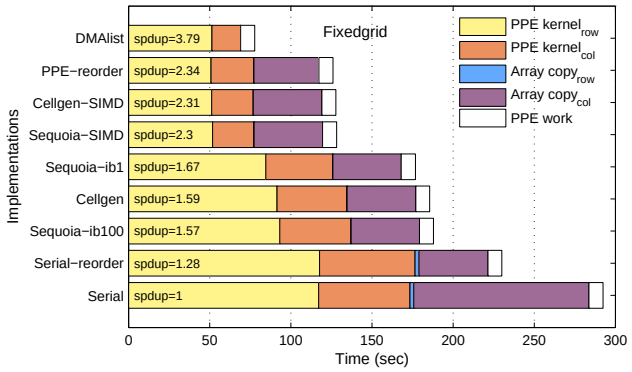
For a quantitative analysis, we compare the performance of each implementation of each application. The experimental environment is a Sony PlayStation 3 running Linux with a 2.6.24 kernel and Cell IBM SDK 3.0. On a PS3 running Linux, only six SPEs are available. The compiler is gcc 4.1.2 with full optimizations turned on. Each data point, except for CellStream, represents the best of 40 runs; we found this more reproducible and representative than the average. For CellStream, the average is more appropriate.

5.1 CellStream

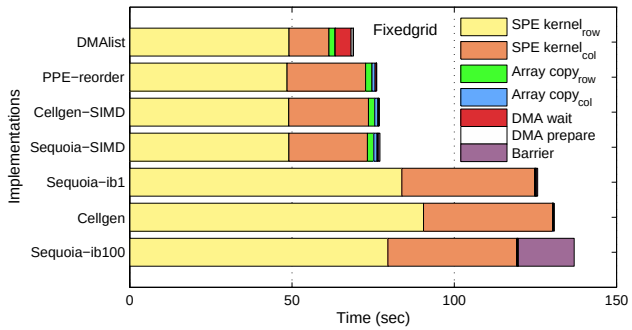
Table 3 shows the peak bandwidth achieved by the Cellgen and Sequoia programming models, a hand written version using the Cell SDK directly, and the PPE only version. The test streamed a 192 MB file cached in memory through a single SPE in 16 KB chunks (except in the PPE only version). The bandwidth was calculated by dividing the total data transferred by the amount of time spent working. Any extra time spent by the PPE threads reading in and writing to disk was not factored in. The PPE only version ran solely on the PPE and modified the memory by using a memset call on the data. It serves as a baseline for comparison, as it represents what can be achieved without the aid of the SPEs.

Table 3 indicates the bandwidth of streaming the data from main memory to the SPE and back to main memory. Data streaming requires a DMA get, some processing, and a DMA put command. The bandwidth calculations in Table 3 are the total data transferred divided by the time the SPE was busy. This SPE busy time includes reading and writing of the data, so each 16 KB of data had two DMA operations performed on it. In order to get the average bandwidth per DMA operation, the bandwidth numbers listed in Table 3 would have to be doubled.

Sequoia and Cellgen both achieve nearly equal results as the hand written version. This is to be expected as all three implementations issued 24,576 DMA calls of 16 KB each (12,288 reads and 12,288 writes). The only difference between the hand written version and the models is the use of double buffering with DMA fencing calls in the former and triple buffering in the latter. The fencing calls used in the hand written version have shown a slight speed up compared to using an extra buffer.



(a) PPE timing profile. *PPE kernel* measures the time to complete each offloaded function. *Array copy* is the time spent on copying arrays from main memory for each discretization function. *PPE work* includes array initialization and file I/O time.



(b) SPE timing profile shows the time breakdown of offloaded functions. *SPE kernel* is the time spent on core computation excluding DMA and array copying overheads. *Array copy* is the time spent on copying arrays to SPE local storage. *DMA wait* is the DMA data transfer time not overlapped with computation in addition to the time for checking the completion of DMA commands. *DMA prepare* is the time to prepare DMA addresses and lists in addition to the time to queue DMA commands.

Figure 3: Performance of each Fixedgrid implementation.

The main bottleneck in all implementations is the Memory Interface Controller. The MIC can access the memory at a rate of 16 bytes per bus cycle. The EIB operates at 1.6 GHz in our experiments (half of the CPU clock frequency) which gives a theoretical peak speed of 25.6 GB/s. Benchmarks which use optimistic, unsynchronized communication have achieved SPE copy speeds at 30% of peak performance (7.68 GB/s) [16].

5.2 Fixedgrid

We experimented with nine different implementations of Fixedgrid to see how the handling of non-contiguous data transfers affects the performance of the application. These implementations represent the choices for how an implicit model can abstract strided accesses. Currently, neither Cellgen nor Sequoia handles strided access to memory transparently.

Fixedgrid has two types of computational kernels: the row discretization kernel and the column discretization kernel. The former requires row data from a contiguous region of memory, and the lat-

ter requires column data from a non-contiguous region of memory. For each time-step iteration, the former is called twice as much as the latter is. In the serial version, column data is copied to a contiguous buffer as each column is needed by the column discretization kernel running on the PPE. The row/column discretization kernel requires a row/column from three different matrices to compute a result row/column.

The serial-reorder version maintains a transposed copy of each matrix. Therefore, no buffer is used in the column discretization kernel in contrast to the serial version. Instead, the values of each transposed matrix are copied as a whole from the original matrix before the column discretization kernel. They are then copied back as a whole to the original matrix after the computation. The kernel accesses data directly from the transposed matrix. This version benefits from higher locality than the serial version since the transformation from row-major to column-major format is grouped by each matrix. Consequently, the serial-reorder version spends less time copying column data as shown in Figure 3(a). The rest of the implementations—except DMAlist—are based on the serial-reorder version. Therefore, they all spend similar amount of time copying column data on the PPE. There is also a smaller amount of time spent copying row data in both of the serial versions. This operation is replaced by DMA calls in other versions.

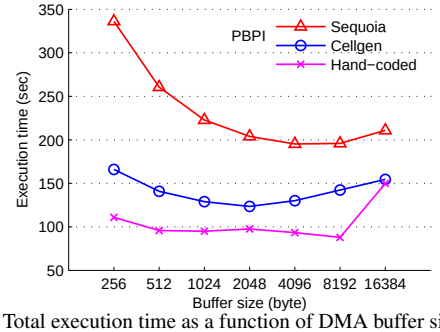
The Cellgen and Sequoia versions are implemented with two types of SPE kernels: conventional operations (as in the serial versions) and SIMD operations. Both of the handed-coded versions, DMAlist and PPE-reorder, are only implemented with the SIMD kernel. The Cellgen and Sequoia versions with non-SIMD kernel show similar performance.

In Sequoia, we test two strategies for mapping data sub-blocks to subtasks, labeled Sequoia-ib100 and Sequoia-ib1. The difference between Sequoia-ib100 and Sequoia-ib1 is the mapping of data sub-blocks to subtasks. Mapping configuration files allow users to specify the interblock mapping strategy that Sequoia uses to decide how data is distributed to subtasks. When the interblock option is set to 100 (Sequoia-ib100), Sequoia performs a block distribution of data with a block size of 100; task_0 takes from block_0 to block_{99}, task_1 takes from block_{100} to block_{199}, etc. When the interblock option is set to 1 (Sequoia-ib1), Sequoia performs an interleaved assignment of data to tasks with a block size of 1; task_0 takes block_0, block_6, block_{12}, task_1 takes block_1, block_7, block_{13}, etc., given that there are six subtasks for six SPEs.

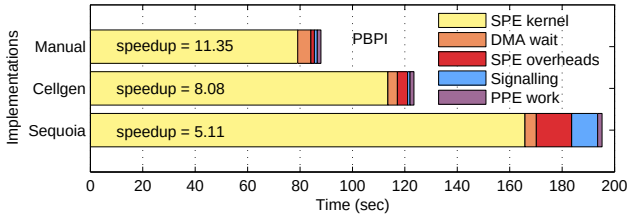
The Cellgen version assigns blocks of contiguous iterations to each SPE. The block size is determined at runtime so that each SPE gets as close to an equal amount of work as possible. This division of work is similar to the Sequoia version when the interblock mapping strategy is 100. It exhibits load imbalance due to the data dependency in computation cost, as shown in Figure 3(b).

The DMAlist implementation of Fixedgrid uses DMA lists to transfer columns of data. DMA lists are the only mechanism provided by the Cell to perform scatter/gather operations. Column accesses are achieved by constructing lists of DMAs for each element in the column. However, since the minimum size of a DMA is 16 bytes, and each element is an 8 byte floating point value, DMA lists transfer unnecessary data. The SIMD operations also work on the data that was transferred as an artifact of the minimum DMA size. Unlike the other versions, DMAlist does not require column data to be reordered on the PPE or SPE.

In the PPE-reorder version, to make the same kernel work on non-interleaved row data, it is obtained by a DMA transfer and is interleaved into a vector array twice as large as the data itself. This copy operation on SPUs introduces the row array copy overhead shown in Figure 3(b). Since columns also require copying—they are reorganized into contiguous arrays—both row and column discretization kernels rely on copying operations.



(a) Total execution time as a function of DMA buffer size.



(b) Comparison of the best cases from each implementation. *SPE kernel* accounts for the time to run hand-coded or generated SPE kernel. *DMA wait* is the DMA data transfer cost that is not overlapped with computation in addition to the time for checking the completion of DMA commands. *SPE overheads* accounts for DMA preparation, barriers, and other programming model specific overheads which vary depending on their implementations. *Signaling* accounts for overhead from signaling between PPE and SPE.

Figure 4: Performance of each PBPI implementation using six SPEs.

The computation of an array element in the Fixedgrid kernel is dependent on the preceding elements in the array. Therefore, it is not possible to further utilize vector operations by condensing a vector array. However, it is possible to utilize wasted vector operation cycles if two row/column elements are fetched and computed at the same time. When a row result is ready in an interleaved array, it is copied back to a non-interleaved buffer array on the local storage for bulk DMA transfer. Cellgen and Sequoia versions with the SIMD kernel rely on the same data rearrangement strategy as that of the PPE-reorder version. Overall, we find that the lack of support for automatic generation of DMA scatter/gather operations is the key reason for the performance gap between the high-level programming models and the hand-tuned version of Fixedgrid.

5.3 PBPI

Applications with a fine granularity of parallelism are sensitive to the size and frequency of DMAs between the SPE and main memory. Since PBPI is such an application, we experimented with different buffer sizes, as shown in Figure 4(a).

With the manual implementation, the optimal performance of PBPI was achieved with a buffer size of 8 KB. The best Sequoia performance was with a buffer size of 4 KB. With Cellgen, the best performance was achieved with a buffer size of 2 KB which is used in 64 unrolled iterations of a computational loop.

The principle behind loop unrolling on the SPE is to maximize the overlap of computation and communication. As the unroll factor increases, so does the amount of data transferred for each DMA. If the size of the DMA is too small, the data transfers can not keep up with the computation. But, as the unroll factor increases, so too does the code size, and eventually the code size becomes too large for the SPE. The best unrolling factor balances DMA size, computation time and code size. Cellgen programmers control loop unrolling by explicitly setting the unroll factor in the directive. Cell-

gen uses this unroll factor to choose a buffer size based on the use of the array inside the loop. These experiments are the groundwork towards deducing the best unroll factor at compile time.

The major factors that influence performance in all three cases are the performance of the computational kernel which is either manually written or generated for the SPE; the overhead of DMA related operations; the extra overheads on SPEs generated by the programming model runtime; and the overhead of signaling between PPE and SPE, as shown in Figure 4(b).

The SPE computational kernel generated by Sequoia relies on data structures to describe the array organization. Array accesses incur overhead due to the additional computation needed to translate the programmer's intent to Sequoia's data layout. This overhead is an instance of a programming model abstraction impacting performance. Similar overheads specific to Sequoia include the constraint checking for the size and alignment of DMA data and the DMA buffer padding to satisfy the constraints.

There are two differences between the SPE computational kernel generated by Cellgen and the computational kernel from the reference code: loop unrolling and a modulus operation introduced to each array access to translate a main memory address to an SPE buffer address. The hand-coded kernel is loop-unrolled and vectorized. The total execution time and the SPE kernel time of each implementation are shown in Figures 5(a), 5(b), and 5(c).

In Cellgen, the iterations are distributed to the SPEs once, before the computation starts, as opposed to dynamically on demand as the SPEs complete iterations. If the distribution of iterations is imbalanced, there may be variance in the time it takes for a single SPE to complete its iterations. The SPE that takes the longest holds up the rest of the computation. The imbalance is a result of using the prescribed buffer size as the atomic unit upon which to divide iterations. As the buffer size increases, the SPE with the most work can have proportionally increasing work. This is the reason that the minimum and maximum SPE kernel time diverge for Cellgen, as shown in Figure 5(b).

Sequoia DMA performance benefits from the redundant copy elimination strategy, which avoids unnecessary DMAs when the same data is used in multiple locations. Cellgen obtains a similar benefit for data declared as private; the data is DMAed to each SPE only once.

The multi-buffering DMA scheme is used to hide DMA data transfer overhead in all three implementations. However, DMA transfer overhead is exposed when waiting for the completion of the DMA command at the beginning and at the end of the iterations, where there is no computation to overlap with. This overhead becomes more pronounced as the buffer size increases. On the other hand, when computation completely overlaps the data transfer, the major overhead is the cost of checking the completion of a transfer, which decreases as the DMA buffer size increases and the number of DMA calls decreases. When the transfer time of a DMA becomes larger than the computation time for an iteration, it cannot be hidden completely and is exposed as overhead. The DMA wait overhead shown in Figures 5(d), 5(e), and 5(f) includes the cost of checking the completion of DMA transfers and their exposed overhead. The DMA prepare overhead includes the cost of issuing DMA commands and the cost of manipulating buffer addresses and DMA tags.

To optimally run an application like PBPI, it is important to balance the DMA data transfer and computation costs. The cross-over point is reached with different buffer sizes in the three implementations. This difference is due to the variance in the execution time for an iteration in each version. The variance exists because each programming model provides different abstractions which have different associated overheads.

The DMA wait overhead becomes minimal when the buffer size is 2 KB in the hand-coded case, while it becomes so at 4 KB for Cellgen and 8 KB for Sequoia. This discrepancy is due to the difference in the cost of the generated computational kernel and data transfer strategies. Optimal performance is achieved when the sum of the computation costs and all related data transfer overheads is minimal. All data transfer overheads include the exposed wait time for DMAs, time spent to prepare DMAs, and time spent to verify DMA completion. This can be seen in Figure 5, as the best performance for each implementation is achieved when the sum of computation costs (*SPE kernel*) and the exposed data transfer overheads (*DMA wait* and *DMA prepare*) are at their minimum. This minimum occurs for the hand-written version at 8 KB, at 2 KB for Cellgen, and at 8 KB for Sequoia.

In the hand-coded case, the epilogue (which includes the computation and communication for the final iterations which are not a multiple of the buffer size) is inefficient: one DMA is issued for each iteration. In Cellgen and Sequoia, one DMA command is issued for the entire remainder of the data. SPE overheads incurred from each DMA transfer decrease as the buffer size increases, and overheads related to an SPE parallel region (such as barriers and signaling) decrease as the total number of parallel regions are called at runtime.

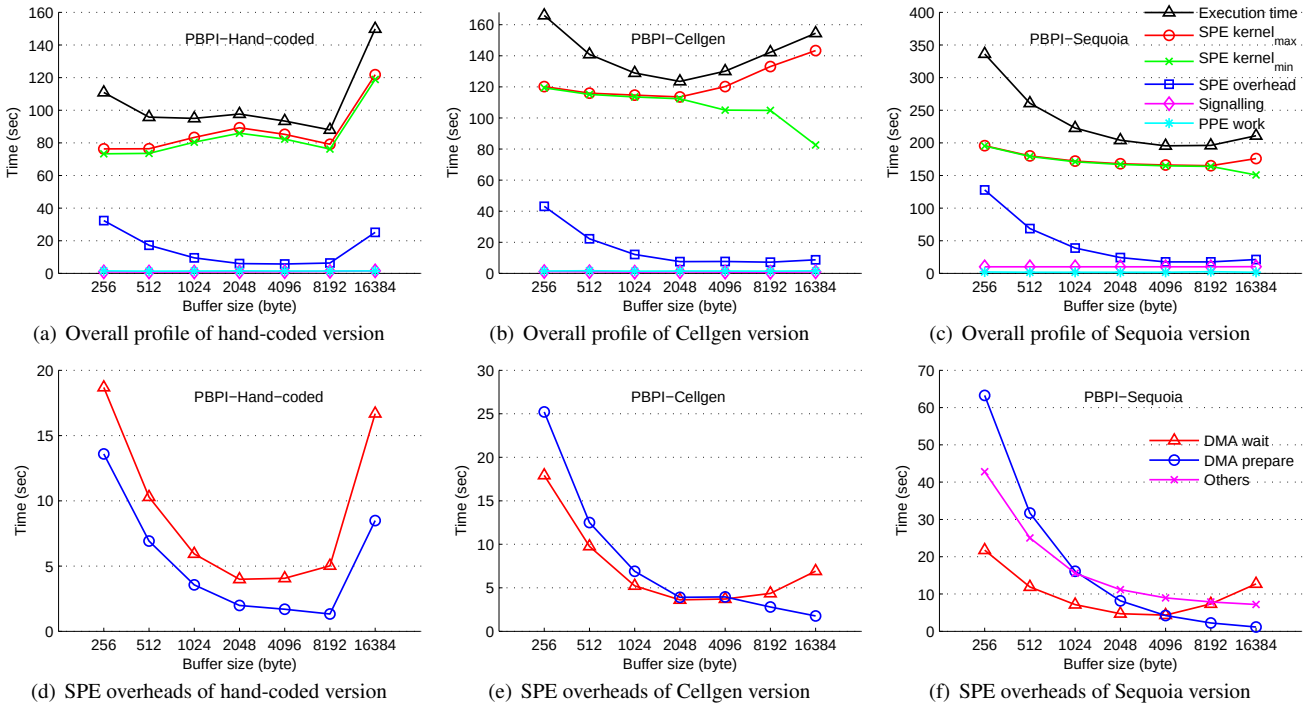
Sequoia has other overheads on the SPE, shown in Figure 5(f), including barriers, reductions, and extra copies of scalar variables which are artifacts of the Sequoia compilation process. Such overheads become noticeable when there is a large number of offloaded function calls. There are 324,071 offloaded function calls in a PBPI run, while there are only 2,592 and 12 offloaded function calls in Fixedgrid and CellStream respectively.

At the end of a leaf task, Sequoia sometimes requires the SPEs to synchronize among themselves for a barrier. In contrast, Cellgen does not require such a barrier among SPEs. Instead, each SPE waits until all outstanding DMAs have completed and then sets a status value in its local storage to indicate completion. The PPE polls these values in each SPE directly, waiting for all SPEs to complete. Cellgen relies on a similar method for collecting the result from SPEs for reduction operations, while Sequoia relies on the DMA and barriers among SPEs. For PBPI, the current Cellgen reduction method is efficient because the reduction data is a single variable. In cases where multiple values are reduced, however, the Sequoia method of using SPEs for reduction operation might be superior. The signaling method used is also different: Sequoia relies on the mailbox communication protocol provided by the Cell SDK, while Cellgen accesses the memory directly. The direct access generally performs better.[1]

6. Related Work

Our study broadly covers two classes of programming models for accelerator-based multi-core processors with explicitly managed memory hierarchies. The first class uses data annotations to specify the data read and written by any given task scheduled to execute on an accelerator. Programming models such as Cell SuperScalar [3], RapidMind [19] and ALF [7] use a high-level task specification with input, output and in/out data attributes. These programming models differ primarily in the internal representation and management of tasks and data transfers. The IBM ALF programming framework automates double buffering using an internal decision tree. The decision tree specifies how many buffers to use (four, three or two) based on the size of the in/out data in each task. Cell SuperScalar automates double buffering through the use of task bundles. Users specify a group of tasks, and the execution

[1] We have observed that mailbox communication performs 20 times slower than DMAs with Linux kernel 2.6.23 and 3 times slower with 2.6.24.



(a) Overall profile of hand-coded version (b) Overall profile of Cellgen version (c) Overall profile of Sequoia version

(d) SPE overheads of hand-coded version (e) SPE overheads of Cellgen version (f) SPE overheads of Sequoia version

Figure 5: The impact of DMA buffer size on the performance of each PBPI implementation. *SPE kernel*$_{\max}$ and *SPE kernel*$_{\min}$ show the maximum and the minimum time spent by SPE kernels among 6 SPEs respectively. The total runtime is bounded by the sum of the maximum kernel time and other overheads.

of a given task in the group is overlapped with the input transfer of a following task and the output transfer of a preceding task. In both ALF and Cell SuperScalar, overlapping is achieved through explicit specification of the tasks and their working sets, rather than through implicit decomposition of a large task and its data set into pieces, which is the case with Cellgen.

Cellgen derives from OpenMP [18] and similar programming models based on the abstraction of a shared address space. Recent efforts for extending OpenMP with directives that manage dependent tasks [9] are directed towards improving locality by automatically managing dependencies (data transfers), between tasks executing on different processors or accelerators. Cellgen takes a different path, by managing locality and communication overlap through implicit task and data decomposition. Streaming languages [8, 12, 13] also expose data locality to the programmer via the stream abstraction. Decomposing data streams into in/out blocks and buffering these blocks in local memories is the equivalent of decomposing loops into tasks and scheduling the transfers for the in/out sets of each task in Cellgen. Earlier studies on streaming languages for both conventional and streaming processor architectures [13, 8] have demonstrated that the stream abstraction enables locality optimization via compiler/runtime support. A similar argument is made in this paper for Cellgen, promoting programmability without performance penalty.

7. Conclusions

Through the study of three programming models of varying complexity, we have shown that implicit management of both parallelism and locality can produce code with performance comparable to that of hand-tuned code and code generated from explicit management of locality. Generating such code requires adequate compiler and runtime support, but it also reduces the programming effort as measured by lines of code.

We have demonstrated this point with Cellgen, a programming model which uses private/shared data classification clauses as the sole mechanism for managing locality. In Cellgen, performance optimization through user intervention is required only to properly manage the granularity of parallel tasks and the distribution of work and data between cores. Both these optimizations are implemented implicitly through parameters passed to directives and do not add substantial programming effort. Furthermore, the compiler and runtime system can be extended to integrate more scheduling algorithms—such as dynamic, interleaved, or work stealing—to further ease the task of the programmer in managing granularity and scheduling.

As expected, we have shown the sensitivity of programming models to data transferring overheads. Each programming model imposes additional specific overheads to each data transfer, and the implementation needs to mask both the endemic overheads of the programming model and the actual data transfer overheads with proper scaling and distribution of the computation. Bulk DMA transfers and elimination of extraneous DMAs for point-to-point and global synchronization improve the performance of high-level programming models, provided that the compiler analysis is powerful enough to perform these optimizations.

This work has certain limitations that we intend to address in future research. Although we selected two programming models that we believe represent the broader spectrum of models proposed for explicitly managed memory hierarchies, our evaluation is not exhaustive. In particular, it is possible that programming models with explicit task and locality management could outperform implicit programming models with applications where statically summarizing and analyzing data access patterns is difficult. Irregular applications such as molecular dynamics simulations may exhibit this problem. The problem is also well documented by the parallelizing/optimizing compiler literature [20]. We believe that a high-level specification of the input/output data sets of tasks is a good starting

point to simplify the analysis required to schedule irregular data transfers through the compiler. Models like Cellgen, which provide higher-level data specification clauses, may face difficulties in implementing certain data transfer optimizations. Further research is required in this space.

In terms of applications studied, we did not investigate applications with inherent load imbalance; the imbalances in our experiments are artifacts of the runtime environment. Inherently imbalanced applications would require further support for dynamic scheduling of tasks in both programming models used in this study. An interesting question here is if the scheduling algorithms that are typically readily available by programming models for use—such as static, dynamic, and guided in OpenMP—need to be revised in view of the specific characteristics of heterogeneous multi-core architectures. In the Cell for example, we have found that setting the default block size for loop iterations and data is not obvious and may incur significant load imbalance, even in theoretically perfectly balanced loops.

We did not investigate applications with irregular and unpredictable data access patterns, where the potential for prefetching and data buffering is limited because of the lack of compile time and runtime information about future data accesses. Such applications raise challenges for both homogeneous cache-based and heterogeneous multi-core architectures with software-managed local storage, and investigating the relative performance of the two platforms remains an open question. Furthermore, we made no effort to expose more than one layers of task and data parallelism across the cores through the programming models, although several applications might benefit from such an approach. We are exploring these issues in ongoing work.

Acknowledgments

This research is or was supported by grants from NSF (CCR-0346867, CCF-0715051, CNS-0521381, CNS-0720750, CNS-0720673), the U.S. Department of Energy (DE-FG02-06ER25751, DE-FG02-05ER25689), IBM, and Virginia Tech (VTF-874197).

References

[1] A. M. Aji, W. Feng, F. Blagojevic, and D. S. Nikolopoulos. CellSWat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine. In *Proceedings of the 2008 ACM Conference on Computing Frontiers (CF08)*, pages 13–22, 2008.

[2] J. Balart, M. González, X. Martorell, E. Ayguadé, Z. Sura, T. Chen, T. Zhang, K. O'Brien, and K. M. O'Brien. A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor. In *Proc. of the 20th International Workshop on Languages and Compilers for Parallel Computing, LNCS Vol. 5234*, pages 125–140, Oct. 2007.

[3] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 86, 2006.

[4] W. P. L. Carter. Documentation Of The Saprc-99 Chemical Mechanism For Voc Reactivity Assessment. Final Report Contract No. 92-329, California Air Resources Board, May 8 2000.

[5] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine and Its First Implementation – A Performance View. *IBM Journal of Research and Development*, 51(5):559–572, Sept. 2007.

[6] T. Chen, Z. Sura, K. M. O'Brien, and J. K. O'Brien. Optimizing the Use of Static Buffers for DMA on a CELL Chip. In *Languages and Compilers for Parallel Computing, 19th International Workshop (LCPC)*, pages 314–329, 2006.

[7] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating Computing With the Cell Broadband Engine Processor. In *Proceedings of the 2008 ACM Conference on Computing Frontiers (CF08)*, pages 3–12, 2008.

[8] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with Streams. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing (Supercomputing'2003)*, page 35, 2003.

[9] A. Duran, J. M. Perez, E. Ayguade, R. M. Badia, and J. Labarta. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In *OpenMP in a New Era of Parallelism, Proceedings of the 4th International Workshop on OpenMP, LNCS Vol. 5004*, pages 111–122, July 2008.

[10] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 83, 2006.

[11] X. Feng, K. W. Cameron, and D. A. Buell. PBPI: A High Performance Implementation of Bayesian Phylogenetic Inference. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 75, 2006.

[12] M. I. Gordon, W. Thies, and S. P. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[13] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: Programming General-Purpose Multicore Processors Using Streams. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–307, 2008.

[14] W. Hundsdorfer. Numerical Solution of Advection-Diffusion-Reaction Equations. Technical report, Centrum voor Wiskunde en Informatica, 1996.

[15] IBM Corporation. Software development kit for multi-core acceleration version 3.0. Oct. 2007.

[16] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez. Performance Analysis of Cell Broadband Engine for High Memory Bandwidth Applications. *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 210–219, April 2007.

[17] J. C. Linford and A. Sandu. Optimizing Large Scale Chemical Transport Models for Multicore Platforms. In *Proceedings of the 2008 Spring Simulation Multiconference*, Ottawa, Canada, April 14–18 2008.

[18] T. Mattson. Introduction to OpenMP – Tutorial. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 209, 2006.

[19] M. D. McCool and B. D'Amora. Programming using RapidMind on the Cell BE – Tutorial. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 222, 2006.

[20] N. Mitchell, L. Carter, and J. Ferrante. Localizing Non-Affine Array References. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 192–202, 1999.

[21] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 95(6):879–899, May 2008.

[22] B. Rose. Cellstream. http://www.cs.vt.edu/~bar234/cellstream.

[23] A. Sandu, D. Daescu, G. Carmichael, and T. Chai. Adjoint Sensitivity Analysis of Regional Air Quality Models. *Journal of Computational Physics*, 204:222–252, 2005.

[24] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 156–166, 2007.