Parallel Programming Models for Heterogeneous Multi-Core Architectures

https://doi.org/10.1109/MM.2010.94

Published in:
IEEE Micro
PARALLEL PROGRAMMING MODELS FOR HETEROGENEOUS MULTICORE ARCHITECTURES

This article evaluates the scalability and productivity of six parallel programming models for heterogeneous architectures, and finds that task-based models using code and data annotations require the minimum programming effort while sustaining nearly best performance. However, achieving this result requires both extensions of programming models to control locality and granularity and proper interoperability with platform-specific optimizations.

Vendors have commoditized heterogeneous parallel architectures through single-chip multicore processors with heterogeneous and asymmetric cores (for example, IBM Cell, AMD Fusion, and NVIDIA Tegra) and through stand-alone wide single-instruction, multiple data (SIMD) and single-instruction, multiple-thread (SIMT) processors that serve as board-level computational accelerators for high-performance computing nodes (such as NVIDIA GPUs and Intel’s Larrabee and Graphics Media Accelerator). Amdahl’s law of the multicore era suggests that heterogeneous parallel architectures have more potential than homogeneous architectures to accelerate workloads and parallel applications.1

Developing accessible, productive, and scalable parallel programming models for heterogeneous architectures is challenging. The compiler and runtime environment of a programming model for heterogeneous architectures must manage multiple instruction-set architectures (ISAs), multiple address spaces, software-managed local memories, heterogeneous computational power, memory capacities, and communication and synchronization mechanisms. Many vendor and academic efforts attempt to address these challenges. IBM advocates the use of OpenMP to exploit parallelism in the heterogeneous Cell processor.2 Vendors have agreed to provide OpenCL on their heterogeneous platforms. Several research environments are also being used, including Sequoia,3 RapidMind (http://www.rapidmind.com), Offload,4 Star Superscalar (Stars),5 and CellGen.6

Efforts seem to converge in at least two common patterns for programming heterogeneous multicore processors. The first is the task-offloading pattern, whereby one or more cores of the processor, preferably those cores that run the operating system and support a general-purpose ISA (for example, x86 and...
PowerPC), offload work to the other cores, which typically have special acceleration capabilities, such as vector execution units and fine-grained multithreading. The second pattern is the explicit management of locality, via control of data transfers to and from the accelerator-type cores, and via runtime support for scheduling these data transfers so they can be overlapped with computation.

Despite convergence, little is known about these models’ relative performance on a common hardware platform and productivity under a common set of applications and assumptions regarding programmers. The literature lacks insights on the importance of parallel programming constructs and the implications of using these constructs on programmer productivity. Furthermore, the literature lacks insight on the relationship between performance and productivity within parallel programming models. To bridge this knowledge gap, we evaluate six programming models designed for and implemented on the Cell processor: Sequoia, StarSs, Cell-Gen, Tagged Procedure Calls (TPC), CellMP (both TPC and CellMP were developed in the context of the SARC project), and IBM’s low-level Cell software developer’s kit (SDK). Our evaluation departs from prior work in that it considers simultaneously performance and productivity in these models, using rigorous metrics on both counts. We evaluate the models using the same hardware, same software stack running beneath the programming models, same applications (see the “Applications” sidebar for a description of the three applications used in our study), and programmers with similar skills and backgrounds (advanced graduate students with extensive exposure to parallel programming and parallel code optimization).

Programming models under study

We make a first attempt at classifying programming models for heterogeneous multicore processors in terms of how they express parallelism and locality (directives versus language types versus runtime library calls), the vehicles of parallel execution (such as tasks and loops), their scheduling scheme (for example, static or dynamic), their means for controlling and optimizing data transfers, and the availability of compiler support for automatic work outlining. Table 1 summarizes this classification.

Hand coding on the cell

Programming the Cell by hand requires using IBM’s Cell SDK 3.0. The SDK exposes Cell architectural details to the programmer, such as SIMD intrinsics for synergistic processor unit (SPU) code. It also provides libraries for low-level, Pthread-style thread-based parallelization, and direct memory access (DMA) commands based on a get/put interface for managing locality and data transfers.

Programming in the Cell SDK is similar in complexity to programming with an explicit remote DMA-style communication interface on a typical cluster7 and with an explicit multithreading interface (such as Posix threads) on a typical multiprocessor. The programmer needs both deep understanding of thread-level parallelization and deep understanding of the Cell processor and memory hierarchy.

Although programming models can transparently manage data transfers, the Cell SDK requires the programmer explicitly identify and schedule all data transfers. 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. Although tedious, 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.

Sequoia

Sequoia expresses parallelism through explicit task and data subdivision. The programmer constructs trees of dependent tasks
in which the inner tasks call tasks further down the tree, eventually ending in a leaf task, which is typically where the real computation occurs. At each level, the data is decomposed and copied to the child tasks as specified. Each task has a private address space.

Sequoia strictly enforces locality because tasks can only reference local data. In this manner, there can be a direct mapping of tasks to the Cell architecture, where the SPU local storage is divorced from the typical memory hierarchy. By providing a programming model in which tasks operate on local data, and providing abstractions to subdivide data and pass it on to subtasks, Sequoia can completely abstract away the underlying architecture from programmers. Sequoia lets programmers explicitly define data and computation subdivision through a specialized notation. Using these definitions, the Sequoia compiler generates code that divides and transfers the data between tasks and performs the computations on the data as described by programmers for the specific architecture. The mappings of data to task and task to hardware are fixed at compile time.

Sequoia relieves programmers from awareness of the underlying architectural data-transfer mechanism between processors and memories by providing a custom memory-allocation API, through which the programmer can reserve memory spaces that meets the alignment constraints of the architecture. The current Sequoia runtime does not support noncontiguous data transfers.

Star Superscalar

StarSs is a task-based parallel-programming model. Similarly to code in OpenMP 3.0, the code is annotated with pragmas, although in this case the pragmas annotate when a function is a task. Additionally, pragmas indicate the direction and access properties (input, output, or inout) of the function’s parameters, with the objective of giving hints to the runtime. This lets the StarSs runtime system automatically discover the actual data dependencies between tasks. For this article, we used the Cell Superscalar (CellSs) instance of StarSs, tailored to the Cell broadband engine (Cell BE) platform.

CellSs is based on a source-to-source compiler and a runtime library. The source-to-source compiler can separate the code corresponding to the PowerPC processing unit (PPU) and to the SPU. It also generates the necessary glue code and the corresponding calls to the runtime library. CellSs then compiles and links this generated code with the IBM SDK toolchain.

The runtime library exploits the existing parallelism by building a task-dependency graph at runtime. The graph parallelism is further increased through renaming. This technique replicates intermediate data, eliminating false dependencies. The runtime system issues data transfers before issuing the task needing the data, letting us overlap computation with communication. Task scheduling is centralized; the runtime system groups in a single bundle several

Table 1. Qualitative properties of the Cell broadband engine programming models evaluated in this study.

<table>
<thead>
<tr>
<th>Model</th>
<th>Concurrency constructs</th>
<th>Parallel execution constructs</th>
<th>Scheduling scheme</th>
<th>Locality control</th>
<th>Automatic function outlining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell SDK</td>
<td>Library calls</td>
<td>Threads</td>
<td>Static/dynamic</td>
<td>Explicit (direct memory accesses, buffers)</td>
<td>No</td>
</tr>
<tr>
<td>CellGen</td>
<td>Directives</td>
<td>Parallel loops/tasks</td>
<td>Static</td>
<td>Implicit (compiler)</td>
<td>Yes</td>
</tr>
<tr>
<td>CellMP</td>
<td>Directives</td>
<td>Parallel loops/tasks</td>
<td>Static</td>
<td>Explicit (copy in/out directives, blocking)</td>
<td>Yes</td>
</tr>
<tr>
<td>Sequoia</td>
<td>Language types</td>
<td>Tasks</td>
<td>Static</td>
<td>Explicit (language in/out data types)</td>
<td>No</td>
</tr>
<tr>
<td>StarSs</td>
<td>Directives</td>
<td>Tasks</td>
<td>Dynamic</td>
<td>Explicit (in/out clauses)</td>
<td>No</td>
</tr>
<tr>
<td>Tagged</td>
<td>Library calls</td>
<td>Tasks</td>
<td>Static</td>
<td>Explicit (argument tags)</td>
<td>No</td>
</tr>
</tbody>
</table>

...
Applications

We use three applications for our analysis: a memory bandwidth benchmark and two realistic supercomputer-class applications.

CellStream

CellStream is a memory bandwidth benchmark that attempts to maximize the rate at which synergistic processor units (SPUs) can transfer data to and from main memory. The benchmark is designed so that a small computational kernel can be dropped in to perform work on data as it streams through SPUs. If no computational kernel is used, the benchmark can match the performance of the read/write direct memory access (DMA) benchmark bundled with the Cell SDK 3.0.

For our experiments, we use an input/output stream of 192 Mbytes of data flowing across the SPUs. Data originates from a file, is read in by a PowerPC processing unit (PPU) thread, transferred in and out of one of the participating SPUs, and written by another PPU thread onto a separate file.

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. It uses an implicit Rosenbrock method to integrate a 79-species SAPRC-99 atmospheric chemical mechanism for volatile organic compounds (VOCs) and nitrogen oxides (NOx) on every grid point. Scientists can selectively disable chemical or transport processes to observe their effect on monitored concentrations.

To calculate mass flux on a 2D domain, FixedGrid calculates a two-component wind vector, horizontal diffusion tensor, and concentrations for every species of interest. 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 \( N_x \) is the domain’s width and \( N_y \) is the domain’s height. Concentration data is stored in an \( N_x \times N_y \times N_L \) array, where \( N_L \) is the number of monitored chemical species. To calculate ozone \((O_3)\) concentrations on a \( 600 \times 600 \) domain as in our experiments, FixedGrid calculates approximately 1,080,000 double-precision values (8.24 Mbytes) at each time step, using 25,920,000 double-precision values (24.7 Mbytes) in the calculation.

PBPI

PBPI is a parallel implementation of the Bayesian phylogenetic inference method, which constructs phylogenetic trees from DNA or amino acid sequences using a Markov chain Monte Carlo (MCMC) sampling method. Two factors determine the computation time of a Bayesian phylogenetic inference based on MCMC: 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. PBPI developers can reduce the length of the Markov chains by developing improved MCMC strategies to propose high-quality candidate states and to improve acceptance/rejection decisions. We can accelerate the computation time per generation 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. Three computational loops are called a total of 324,071 times and account for most of the program’s execution time. The first loop accounts for 88 percent of the calls and requires 1.2 Mbytes to compute a result of 0.6 Mbytes. The second loop accounts for 6 percent of the calls and requires 1.8 Mbytes to compute a result of 0.6 Mbytes. The third also accounts for 6 percent of the calls and requires 0.6 Mbytes to compute the weighted reduction of a vector onto a result of 8 bytes.

References

identify parallel sections of their code in the form of loops accessing particular segments of memory. Programmers must annotate these sections to mark them for parallel execution, and 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. Note that although the data set accessed by each parallel section is annotated, the data set accessed by each task that executes a part of the work in the parallel section is not annotated.

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 SPU 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 must be streamed into the SPU’s local store, out data must be streamed out of local stores, and inout data must 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 manages locality by triggering and dynamically scheduling the associated data transfers.

Being able to stream in, out, and 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 might 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, whereas 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; and 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.

SPUs 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, the compiler schedules loop iterations to SPUs. The current implementation uses static scheduling, whereby the iterations are divided equally among all SPUs.

Tagged Procedure Calls

TPC is a programming model that exploits parallelism through asynchronous execution of tasks. The TPC runtime system creates a task using a function descriptor and argument descriptors as input and identifies the task using a unique task ID. The programmer uses library calls to identify certain procedure calls as concurrent tasks and specify properties of the data accessed by them, to facilitate their transfers to and from local memories. Each argument descriptor is formed by a triplet containing the base address, the argument size, and the argument type, which can be in, out, or inout. The TPC runtime handles contiguous and strided arguments with a fixed stride. For the latter, the programmer masks the type field with the stride flag and packs the number of strides, stride size, and offset within the size field using a macro. The task argument sizes define the granularity of parallelism within a region of straight-line code or a subset of a loop’s iteration space. The programmer implements synchronization using either point-to-point or collective wait primitives.

The PPU issues tasks across SPUs round robin, using remote stores. Each active SPU thread has its own private queue that the issuer can access to post tasks. Each SPU runs a thread that continuously polls its local queue and executes any available tasks in first in, first out (FIFO) order. For each new task, the main thread running on the PPU tries to find an empty slot in some SPU queue to issue the task. Upon task
completion, each SPU thread updates the task status in a thread-specific completion queue located in the PPU cache. In this case, to avoid cache invalidation and the resulting off-chip transfer, the TPC runtime uses atomic DMA commands to unconditionally update the PPU’s cache. PPU polls these completion queues to detect task completion. This polling is done when no more tasks can be issued or a synchronization primitive has been reached.

The TPC runtime uses only on-chip operations when initiating and completing tasks, whereas argument data might require off-chip transfers. The runtime uses task argument prefetching and outstanding write-backs to overlap communication with computation.

CellMP
CellMP uses the OpenMP 3.0 task approach to express the work to be executed in the accelerators. It uses a new clause device (accelerator-kind-list)\(^{10}\) to annotate tasks to be executed on an accelerator. In CellMP, the use of the \texttt{copy\_in} (variable | array-section) clause causes a data transfer from main memory to the accelerator; the \texttt{copy\_out} (variable | array-section) clause will similarly cause the variable or array section to be written back from the accelerator to main memory. The programmer can choose to split the parallel computation to work on chunks of main memory arrays that fit in the local store, given that local stores typically have limited size and cannot fit entire arrays from main memory. To stream large arrays through local memories, a programmer specifies \texttt{array-sections}, as in Fortran90. Communication in CellMP is implemented synchronously. To support communication-computation overlap, CellMP uses CellMT threads.\(^{11,12}\) Usually, two SPU threads are created in each SPU, so that while one of them is computing, the other is in the data-transfer stage.

CellMP also uses loop blocking,\(^{13-15}\) to coarsen the granularity of work and to overcome accelerators’ memory constraints. The scope of blocking includes the \(N\) loops surrounding the enclosed loop body and it is defined by a clause factors (\(F_1, F_2 \ldots F_N\)) to specify the blocking factor of each loop in the considered loop nest, starting from the outermost loop. Code examples showing the use of CellMP directives are available elsewhere.\(^{10}\)

Evaluation
We first evaluate the memory bandwidth achieved by each of the programming models using CellStream and then present the evaluation of two realistic applications, Fixedgrid and Parallel Bayesian Phylogenetic Inference (PBPI) (see the “Applications” sidebar for a description of these applications). The PhD student authors of this article parallelized these applications. They all have similar experience and backgrounds in parallel programming and parallel application optimization on Cell.

We performed the evaluation on QS20 Cell BE blades at the Barcelona Supercomputing Center. Two Cell processors clocked at 3.2 GHz and 1 Gbyte of DRAM power the blades. We installed the IBM Cell SDK 3.0 on each blade. We ran all experiments within the default Linux (version 2.6.22.5-\texttt{fc7}) execution environment, with the default virtual page size of 65,536 bytes (64 Kbytes). We executed each experiment 20 times and calculated means and standard deviations. The machines were used in dedicated mode.

We evaluated the benchmarks with respect to changes in the number of lines needed for the parallelization and exploitation of the Cell BE SPUs. To count lines, we eliminate comments and empty lines, and then use \texttt{indent} so that the code expressed in all programming models is organized with the same rules. The codes are available at http://people.ac.upc.edu/xavim/sarc/pm-codes.tar.bz2.

CellStream
Table 2 shows, for each of the programming models under evaluation, the number of lines we added or removed and the pragmas we added to CellStream to be able to run it using the Cell SPUs as accelerators. We made all changes manually, before compiling the application with the corresponding programming model.

The programmer must add more than 100 lines of code to port the benchmark to
run on the SDK. This code includes thread creation and synchronization on the PPU, explicit DMA transfers on the PPU and SPUs, and management of DMA completion notifications. Sequoia needs half the number of lines added, and TPC reduces them to one third. Sequoia eliminates code for explicit DMA transfers; however, it requires two versions of each function enclosed in a parallel task: a nonleaf version, which is executed on the PPU to partition work between SPUs in leaf tasks and maps these tasks to SPUs; and a leaf version, which encloses the actual task executed on the PUs.

TPC also eliminates code for DMA transfers; however, it requires the programmer to write an SPU task dispatcher for selecting between different offloaded tasks on the SPU and for calculating the number and size of copy in and copy out arguments. This requirement adds 27 lines of SPU code. CellGen, StarSs, and CellMP require one fourth or fewer changes, due to the use of pragmas. The changes for all three models are similar. They consist of using pragmas to annotate the function (StarSs) or the code (CellGen and CellMP) to be offloaded. CellMP allows an additional blocking transformation of copy_in and copy_out data with pragmas, so it usually has more pragmas added than the other models. CellGen is closer to OpenMP, which annotates entire loops for parallelization and therefore incurs the fewest code changes.

Figure 1 shows the bandwidth for CellStream measured in Gbytes per second (GBps). Because memory bandwidth limits the benchmark and Cell blades have two distributed memory modules, we present our evaluation with and without nonuniform memory access (NUMA)-aware memory allocation. The plot clearly shows the benefits of using NUMA-aware memory allocation.

---

**Table 2. Changes in line counts in CellStream.**

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>Change</th>
<th>SDK</th>
<th>Sequoia</th>
<th>CellGen</th>
<th>StarSs</th>
<th>TPC</th>
<th>CellMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td>Offloading</td>
<td>Added</td>
<td>29</td>
<td>7</td>
<td>7</td>
<td>16</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>function</td>
<td>Added</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>spe_main.c</td>
<td>SPE task/kernel</td>
<td>Added</td>
<td>81</td>
<td>46</td>
<td>—</td>
<td>27</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>Total</td>
<td>All lines</td>
<td>Added</td>
<td>110</td>
<td>53</td>
<td>8</td>
<td>18</td>
<td>37</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>User functions</td>
<td>Added One task</td>
<td>One task</td>
<td>—</td>
<td>One task and one leaf</td>
<td>None</td>
<td>None</td>
<td>One task</td>
<td>None</td>
</tr>
</tbody>
</table>
The NUMA-aware versions of CellGen, CellMP, and SDK outperform their NUMA-unaware counterparts, as well as Sequoia, TPC, and StarSs. These runtime systems enumerate SPUs and establish affinity between SPUs and their local DRAM modules, in a way that the memory touched by each SPU is initially allocated in a local DRAM module. StarSs, on the other hand, uses interleaved memory allocation, which places every other page in the same node upon first access to the page. StarSs cannot control the data-to-SPU mapping because of its dynamic scheduler. This, in turn, limits the achieved memory bandwidth. The TPC runtime is unaware of NUMA and suffers from a problem similar to StarSs because of the use of a dynamic task scheduler.

The differences in performance between the NUMA-aware versions of CellGen, CellMP, and SDK runtime systems arise because of a different SPU enumeration and clustering scheme. When the application requests fewer than eight SPUs, the SDK distributes the requested number of SPUs evenly between the blade’s two Cell nodes, whereas CellGen and CellMP cluster the requested SPUs on the same node. The SDK’s SPU-distribution scheme therefore increases the memory bandwidth available to each SPU.

**Fixedgrid**

Table 3 shows the changes we made to Fixedgrid. Our observations about coding complexity in CellStream also hold for Fixedgrid. The Sequoia version uses additional SPE code to improve the performance of DMA transfers. CellGen minimizes the required code changes; however, this happens not only because of the use of pragmas to annotate entire loops, but also because CellGen cannot offload three matrix transpositions to SPUs, which are offloaded in the other programming models. The reason is that the current implementation of strided data accesses in CellGen requires explicit data padding from the programmer. Such data padding makes the resulting matrix incompatible with the format expected by the vectorized kernels. In this case, interoperability between the memory-allocation scheme in the programming model and the underlying machine-specific vectorization framework is essential to maximize performance.

Figure 2a shows the performance of Fixedgrid in terms of the speedup of each

---

**Table 3. Changes in line counts in Fixedgrid.**

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>Change</th>
<th>SDK</th>
<th>Sequoia-Scaler</th>
<th>Sequoia-SIMD</th>
<th>CellGen</th>
<th>StarSs</th>
<th>TPC</th>
<th>CellMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport.c</td>
<td>Offloading functions</td>
<td>Added lines</td>
<td>42</td>
<td>27</td>
<td>27</td>
<td>32</td>
<td>22</td>
<td>95</td>
<td>40</td>
</tr>
<tr>
<td>(103 lines)</td>
<td>Added pragmas</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>5</td>
<td>—</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Removed lines</td>
<td>33</td>
<td>15</td>
<td>33</td>
<td>11</td>
<td>17</td>
<td>14</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>transport_spe.c</td>
<td>SPE task</td>
<td>Added lines</td>
<td>349</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>discretize.c</td>
<td>SPE kernel</td>
<td>Added lines</td>
<td>219</td>
<td>119</td>
<td>314</td>
<td>0</td>
<td>19</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>(143 lines)</td>
<td>Removed lines</td>
<td>60</td>
<td>0</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>All lines</td>
<td>Added</td>
<td>610</td>
<td>146</td>
<td>341</td>
<td>34</td>
<td>46</td>
<td>146</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Removed</td>
<td>93</td>
<td>15</td>
<td>93</td>
<td>11</td>
<td>17</td>
<td>14</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>User functions</td>
<td>Added Two tasks</td>
<td>Two tasks and two leaves</td>
<td>None</td>
<td>Two tasks and two leaves</td>
<td>None</td>
<td>Two tasks</td>
<td>Two tasks</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

---

SEPTEMBER/OCTOBER 2010 49
solution with regard to the time obtained when running the serial version on the PPU (237 seconds). We use the SDK-SIMD version as a reference. The SDK-SIMD and Sequoia-SIMD versions fully vectorize the row discretization performed in Fixedgrid. With this optimization, data transfers and SIMD operations are fully overlapped and optimized, respectively. All programming models scale similarly, except for CellGen, because of the inability to offload the data transpositions. For TPC and CellMP, we use two kernel codes. The gcc compiler automatically vectorizes the gccSIMD version; we hand-tuned the optimized kernel (OPTK) version, eliminating most branches in the kernel’s innermost function. The results show that autovectorization can occasionally achieve comparable performance to manual vectorization and that both TPC and CellMP interoperate efficiently with alternative kernel vectorization schemes, in the sense that their memory-management schemes prevent neither automatic nor hand-tuned vectorization.

PBPI

Table 4 summarizes the changes in source code for PBPI. Our observations on coding effort are similar to the ones we made for the other benchmarks. TPC in particular adds 215 lines of code on the PPU because of the use of a separate offloading code path for each of the three main kernels of PBPI that are enclosed in tasks (labeled L1, L2, and L3). For CellGen-SIMD, StarSs-SIMD, and CellMP-SIMD, we provide separate line counts for each kernel.
Figure 2b shows the performance obtained from each of the programming models in terms of speedup with regard to the serial time obtained in the PPU (671 seconds). It shows the Scalar and SIMD versions for each programming model evaluated, and the hand-coded SDK-SIMD version. SIMD versions get increased performance compared to scalar versions. In PBPI, tasks are small enough to stress the PPU spawning mechanism. We have found two ways to achieve higher performance. On one hand, StarSs and CellMP can eliminate barriers between parallel regions. StarSs does so by using runtime detection of dependences between tasks, and running chains of dependent tasks on the same SPU. CellMP static scheduling of tasks to SPUs allows the programmer to remove the barriers if there is no reduction operation in the parallel loops. On the other hand, CellGen, CellMP, and the SDK versions exploit parallelism in a way similar to OpenMP parallel for loops. Exploiting parallel loops in this way coarsens granularity and creates fewer tasks. Having to create fewer tasks puts less pressure on the runtime system task instantiation and scheduling mechanism that runs on the PPU. Furthermore, each task must bring data to work with at runtime. CellMP achieves this through an option to express copy_in/copy_out data movements in the middle of task code, rather than only at the beginning and the end. Even with this approach, CellMP-SIMD is 4 seconds slower than the SDK version when using 16 SPUs.

Acknowledgments
We thankfully acknowledge the support of the European Commission through the SARC IP project (contract no. 27648), the Encore Strep project (contract no. 248647), and the SEPTEMBER/OCTOBER 2010
the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the MCF IRG project I-Cores (contract no. IRG-224759), the Spanish Ministry of Education (contracts TIN2007-60625 and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980), and the BSC-IBM MareIncognito project; the support of the US National Science Foundation through grants CCR-0346867, CCF-0715051, CNS-0521381, CNS-0720750, and CNS-0720673; the support of the US Department of Energy through grants DE-FG02-06ER25751 and DE-FG02-05ER25689; and the support of IBM through grant VTF-874197.

References

Roger Ferrer is a researcher at the Barcelona Supercomputing Center. His research interests include language and compiler support for parallel programming models. Ferrer has a master’s degree in computer architecture and network systems from the Universitat Politècnica de Catalunya.

Pieter Bellens is a PhD student at the Universitat Politècnica de Catalunya and a member of the Parallel Programming Models group at the Barcelona Supercomputing Center.
Center. His research interests include parallel programming, scheduling, and the cell broadband engine. Bellens has a master’s degree in computer science from the Katholieke Universiteit Leuven, Belgium.

**Vicenç Beltrán** is a senior researcher in the Computer Science Department at the Barcelona Supercomputing Center. His research interests are heterogeneous and hybrid systems, domain-specific languages, and operating systems. Beltrán has a PhD in computer science from the Universitat Politècnica de Catalunya.

**Marc Gonzalez** is an associate professor in the Computer Architecture Department at the Universitat Politècnica de Catalunya (UPC). His research interests include parallel computing, virtualization, and power consumption. Gonzalez has a PhD in computer science from UPC.

**Xavier Martorell** is an associate professor in the Computer Architecture Department at the Universitat Politècnica de Catalunya (UPC). His research interests include support for parallel computing, programming models, and operating systems. Martorell has a PhD in computer science from UPC. He is member of IEEE.

**Rosa M. Badia** is manager of the grid computing and clusters group at the Barcelona Supercomputing Center and a scientific researcher at the Spanish National Research Council. Her research interests include performance prediction and modeling of MPI programs and programming models for complex platforms (from multicore to the grid/cloud). Badia has a PhD in computer science from the Universitat Politècnica de Catalunya.

**Eduard Ayguadé** is a full professor at the Universitat Politècnica de Catalunya and associate director for research in computer sciences at the Barcelona Supercomputing Center. His research interests include multicore architectures, programming models and their architectural support.

**Jae-Seung Yeom** is a graduate student at Virginia Tech. His research interests include parallel programming models and large-scale simulation. Yeom has an MS in information networking from Carnegie Mellon University. He is a member of IEEE.

**Scott Schneider** is a PhD candidate in the Computer Science Department at Virginia Tech and a member of the Parallel Emerging Architecture Research Laboratory. His research interests include high-performance computing, systems, and programming languages. Schneider has a master’s degree in computer science from the College of William and Mary. He is a member of IEEE and the ACM.

**Konstantinos Koukos** received an MS in computer science from the University of Crete. His research interests include support for parallel applications and heterogeneous computing.

**Michail Alvanos** is a PhD student at the Universitat Politècnica de Catalunya. His research interests include parallel programming models and heterogeneous architectures. Alvanos has an MS in computer science from the University of Crete.

**Dimitrios S. Nikolopoulos** is an associate professor of computer science at the University of Crete and an affiliated faculty member of the Institute of Computer Science at the Foundation for Research and Technology-Hellas (FORTH). His research interests include the hardware-software interface of parallel computer architectures. He is a member of IEEE and the ACM.

**Angelos Bilas** is an associate professor at the Institute of Computer Science at the Foundation for Research and Technology-Hellas and the University of Crete. His research interests include architectures and runtime-system support for scalable systems, low-latency high-bandwidth communication protocols, and miniaturization of computer systems. Bilas has a PhD in computer science from Princeton University.

Direct questions and comments about this article to Xavier Martorell, Computer Sciences Dept., Barcelona Supercomputing Center, Campus Nord—C6, c/Jordi Girona 1,3, 08034 Barcelona, Spain; xavim@ac.upc.edu.