Optimizing Large Scale Chemical Transport Models for Multicore Platforms

John C. Linford and Adrian Sandu
Virginia Polytechnic Institute and State University
jlinford@vt.edu, sandu@cs.vt.edu

Keywords: Chemical Transport Model, Domain Decomposition, Time Splitting, Multicore, Scalability, IBM Cell Broadband Engine, Intel Quad-Core Xeon.

Abstract
The performance of a typical chemical transport model is determined on two multicore processors: the heterogeneous Cell Broadband Engine and the homogeneous Intel Quad-Core Xeon shared-memory multiprocessor. Two problem decomposition techniques are discussed: dimension splitting for promoting parallelization in chemical transport models, and time splitting, for reducing truncation error. Additionally, a scalable method for accessing random rows or columns of a matrix of arbitrary size from the accelerator units of the Cell Broadband Engine is presented. This scalable access method increases chemical transport model efficiency by an average of 30% and significantly improves the scalability of dimension-splitting techniques on the Cell Broadband Engine. Experiments show that chemical transport models are 31% more efficient on the Cell Broadband Engine when only six accelerator units are used than on a shared-memory multiprocessor with eight executing cores. Our fully-optimized models achieve an average 118% speedup on the Cell Broadband Engine, and an average 87.5% speedup on a shared-memory multiprocessor with OpenMP.

1. INTRODUCTION
Increasing power consumption, heat dissipation, and other issues have lead to the rapid prevalence of multicore microprocessor chip architectures in recent years. Exploiting the potential parallelism provided by multiple cores on a single chip is an unprecedented challenge to application developers. Since the majority of high performance scientific applications were not designed with multicore architectures in mind, understanding how to port existing applications to new multicore architectures is important.

Comprehensive multiphysics simulations are common in the environmental sciences and life sciences. Air quality models and weather prediction models involve atmospheric simulations which consider dozens of variables such as temperature, pressure, density, terrain features, diffusion tensors, and wind vector fields for every point in the domain. More comprehensive models consider particulates, lightning, variable-height emission sources, and other factors. These models solve systems of equations involving between $10^6$ and $10^7$ double-precision floating point values and require hours or days of runtime on small Beowulf clusters.

Many large-scale multiphysics models contain a high degree of parallelism. Mass flux is typically approximated through a finite difference scheme on a fixed domain grid. Flux can be calculated independently at each grid point or in rows and columns of grid points, depending on the domain decomposition and time stepping scheme. Chemical reactions, photosynthetic processes, and particulate processes are embarrassingly parallel on a fixed grid. Memory I/O, still the major limiting factor in most high performance applications, can overlap computation in machines with large memories and nonblocking I/O support. These concurrent, independent processes make large-scale multiphysics models ideally suited to parallel architectures.

We discuss a simple serial chemical transport model written in Fortran 90 and port it to two multicore processors: the Cell Broadband Engine, an accelerator-based heterogeneous multiprocessor, and the Intel Quad-Core Xeon, a homogeneous shared-memory multiprocessor. We give detailed performance results for the transport model as it computes ozone over a large spatial domain and quantify the benefits of several important optimizations. The model exhibits superior speedup and efficiency on the Cell Broadband Engine which is attributable to the multi-level parallelism provided by the Cell’s accelerator-based heterogeneous architecture.

2. RELATED WORK
There is a growing body of work demonstrating the excellent performance of specialized algorithms on multicore processors. Chen et al. [4] documented the optimization process of a Fast Fourier Transform on the IBM Cyclops-64 chip architecture, a large-scale homogeneous multicore processor, and the Intel Quad-Core Xeon, a homogeneous shared-memory multiprocessor. We give detailed performance results for the transport model as it computes ozone over a large spatial domain and quantify the benefits of several important optimizations. The model exhibits superior speedup and efficiency on the Cell Broadband Engine which is attributable to the multi-level parallelism provided by the Cell’s accelerator-based heterogeneous architecture.
Their method exhibits good scalability on the IBM Cyclops-64 multicore architecture up to 100 cores, showing that enhanced compilers can produce efficient multicore code, even if the programmer lacks domain-specific knowledge. We use both domain-specific knowledge and optimized compilers to achieve good performance on multicore CPUs.

Many high-performance applications show significant performance improvement on multicore processors. Alam and Agarwal [1] characterized computation, communication and memory efficiencies of bio-molecular simulations on a Cray XT3 system with dual-core Opteron processors. They found that the communication overhead of using both cores in the processor simultaneously can be as low as 50% as compared to the single-core execution times. Benthin et al. [2] ported a pure ray tracing application to the Cell Broadband Engine and achieved a performance of nearly one order of magnitude improvement over a single-processor system. By considering a dual-core, quad-thread IBM Power5 processor. Benthin et al. [2] realized a four-times overall performance increase on the Cell in comparison with a dual-processor Hyperthreaded Xeon system and a 5-10% performance increase over a single-processor system with a dual-core, quad-thread IBM Power5 processor.

3. CHEMICAL TRANSPORT MODELING

Chemical transport models (CTMs) solve mass balance equations for concentrations of trace species in order to determine the fate of pollutants in the atmosphere [12]. These equations are described in detail in [13] and [8]. The basic mathematical equations for transport and reaction can be derived by considering these mass balance equations. Let $c'_t = (c'_1(x,t), \ldots, c'_s(x,t))^T$ be concentrations of $s$ chemical species, with spatial variable $x \in \Omega \subset \mathbb{R}^d$ ($d = 2$ or 3), and time $t \geq 0$.

Transport and reaction are given by

$$
\frac{\partial c'_t}{\partial t} = -\sum_{k=1}^{d} \frac{\partial}{\partial x_k} \left( a_k(x,t) c'_k \right) + \sum_{k=1}^{d} \frac{\partial}{\partial x_k} \left( d_k(x,t) \frac{\partial}{\partial x_k} c'_k \right) + f_j(c'_{k,x}, x, t)
$$

(1)

where $a_k$ are the atmospheric velocities, $d_k$ are the diffusion coefficients, and $f_j(c'_{k,x}, x, t)$ describes the nonlinear chemistry of chemical species $s$ together with emissions and deposition.

By considering a simple space discretization on a uniform grid ($x_i = ih$) with a mesh width $h = \frac{1}{m}$, we can approximate solutions to Equation 1 by finite difference methods. This produces the third order upwind-biased advection discretization (Equation 2) and the second order diffusion discretization (Equation 3) for $r' > t$ [9].

$$
c'_t = \begin{cases} 
\frac{a}{h} \left( -\frac{1}{6} c'_{t-2} + c'_{t-1} \right) - \frac{a}{h} \left( \frac{1}{2} c'_{t} - \frac{1}{2} c'_{t+1} \right) & \text{if } a \geq 0 \\
\frac{a}{h} \left( \frac{3}{4} c'_{t-1} + \frac{1}{4} c'_t \right) - \frac{a}{h} \left( \frac{1}{4} c'_{t+1} + \frac{1}{4} c'_{t+2} \right) & \text{if } a < 0
\end{cases}
$$

(2)

$$
c'_t = \frac{(d'_t + d'_{t+1})(c'_{t-1} + c'_t)^T}{2h^2} - \frac{(d'_t + d'_{t+1})(c'_{t} + c'_{t+1})^T}{2h^2}.
$$

(3)

In considering a d-dimensional model, we can apply Equations 2 and 3 to each dimension independently. Thus, we consider the domain in terms of rows and columns of the matrix of chemical concentrations, as shown in Figure 2. This approach is known as dimensional splitting, and has many distinct advantages. By reducing an n-dimensional problem to a set of one-dimensional problems, we introduce a high degree of parallelism. Equations 2 and 3 can be implemented as a single computational routine and applied to each row and column of the concentration matrix individually and in parallel. Dimensional splitting is common in many large scale chemical transport models and other scientific applications.

Applying dimension splitting to these equations introduces a local truncation error at each time step. This error can be reduced by a time splitting method [9]. Time splitting methods are frequently used when applying different time stepping methods to different parts of an equation. For example, chemical processes are very stiff, which calls for an implicit ODE method, but explicit methods are more suitable for space-discretized advection. By interleaving time steps taken in each dimension, we can reduce the truncation error. Figure 1 shows a first order time splitting method for a two-dimensional discretization split into two one-dimensional problems.

3.1. Case Study Application

Our case study application is FIXEDGRID, a two-dimensional, serial, multi-scale chemical transport model written in Fortran 90. FIXEDGRID solves a system of partial differential equations to simulate the transport and diffusion of a chemical plume subject to atmospheric conditions. It can be combined with nonlinear chemical mechanisms to comprehensively model the evolution of an atmospheric state, or used alone to model trace species concentrations. In the experiments, we consider ozone (O$_3$) concentrations on a domain of 1000 $\times$ 700 points mapped to a 250 $\times$ 175 kilometer area, for twelve hours.
The major memory requirements of FIXEDGRID are the concentration matrix, the wind field matrix, and the diffusion matrix. These matrices are all of dimension $NY \times NX$. Like most fixed grid chemical transport models, FIXEDGRID uses dimensional splitting to reduce a two-dimensional problem into a set of one-dimensional problems. The concentration matrix is updated in rows (discretization along the x-axis) and in columns (discretization along the y-axis), so FIXEDGRID naturally stores these matrices as two-dimensional arrays. The concentration matrix stores chemical concentrations for each of the $s$ chemical species for each grid point, totaling $s \times NX \times NY$ double-precision floating point variables. For our experiments, this amounts to 700,000 double-precision variables or approximately 5.34MB of data. The wind field contains two double-precision floating point variables for each grid point, amounting to approximately 10.68MB of data. The diffusion matrix stores one double-precision floating point variable for each grid point, totaling 5.34MB of data. Approximately 22.31MB of data are calculated for each time step. The final output file size is just over 41.1MB.

FIXEDGRID uses first order time splitting to reduce truncation error (see Figure 1). A half time step ($\Delta t/2$) is used when calculating mass flux along the domain x-axis, and a whole time step is used when calculating mass flux along the domain y-axis. This process is equal to calculating the mass flux in one step, but reduces truncation error by $O(h^2)$ [9]. Note that this doubles the work required to calculate mass flux for a full row of the concentration matrix. It is therefore desirable for row discretization to be highly efficient.

The block nature of FIXEDGRID’s domain makes it ideally parallelizable across many CPUs. Distributing a large matrix across many CPUs is a thoroughly-explored problem unaffected by the details of the computational node. Therefore, we consider FIXEDGRID execution on only one node.

4. THE CELL BROADBAND ENGINE

The IBM Cell Broadband Engine is a heterogeneous multicrore processor which has drawn considerable attention in both industry and academia. The Cell was originally designed for the game box market, and therefore it has a low cost and low power requirements. Nevertheless, it has archived unprecedented peak single-precision floating point performance, making it suitable for high-performance computing. IBM announced in February 2006 the development of a petaflop Cell-based system, to be installed in the DOE facility at the Los Alamos National Laboratory in 2008.

The main components of the Cell are a multithreaded Power Processing element (PPE) and eight Synergistic Processing elements (SPEs) [7]. These elements are connected with an on-chip Element Interconnect Bus (EIB) with a peak bandwidth of 204.8 Gigabytes/second. The PPE is a 64-bit dual-thread PowerPC processor with Vector/SIMD Multimedia extensions [10] and two levels of on-chip cache. Each SPE is a 128-bit processor with two major components: a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). All SPE instructions are executed on the SPU. The SPU includes 128 registers of 128 bits and 256 KB of software-controlled local storage. The Cell is primarily a single-precision floating point processor. The Cell’s peak single-precision FP performance is 230.4 Gflops, but peak performance for double-precision is only 21.03 [5]. The next generation of Cell processors is expected to fully pipeline double-precision FP operations.

The only memory directly available to an SPE is its own local storage. The SPE must use direct memory access (DMA) requests to access RAM. The DMA transfers are handled by the Memory Flow Controller. Programmers must be aware that data transferred between local storage and main memory must be 128-bit aligned, the size of each DMA transfer can be at most 16 KB, and the MFC supports only DMA transfer of blocks are are 1, 2, 4, 8, or multiples of 16 bytes long.
5. APPLICATION PORT AND DISCUSSION

This section discusses the implementation, analysis, and optimizations of FIXEDGRID on a shared-memory multiprocessor with OpenMP and the Cell Broadband Engine. The experiments consider ozone (O₃) concentrations on a domain of 1000 x 700 points mapped to a 250 x 175 kilometer area, for twelve hours. The time step ∆t is 50 seconds. Experiments on the Cell were conducted on one node of the the Virginia Tech Sony PlayStation 3 cluster. Each node has one Cell Broadband Engine CPU and 256MB of XDR RAM. Only six of the eight SPE units in each Cell CPU are available to users in the PlayStation 3. Experiments on the multiprocessor were conducted on a Dell PowerEdge SC1430 workstation with two Intel Quad-Core Xeon CPUs with 8GB of DRAM.

5.1. FIXEDGRID on the Cell

Porting FIXEDGRID to the Cell Broadband Engine was nontrivial and resulted in four new versions of the FIXEDGRID code. Each new version was carefully profiled and analyzed to gauge the benefit of the changes we introduced. The changes introduced in each version are:

1. Translate Fortran 90 code to C,
2. Parallelize and offloaded computational cores to the SPEs,
3. Double buffer DMA transfers, unroll loops executed on the SPEs, and vectorize the code via SIMD intrinsics,
4. Implement scalable random matrix column access methods on the SPEs.

Average wall clock runtimes on the Cell Broadband Engine for each version are shown in Table 1.

<table>
<thead>
<tr>
<th>N-SPEs</th>
<th>Ver. 1</th>
<th>Ver. 2</th>
<th>Ver. 3</th>
<th>Ver. 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2638.04</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>3965.45</td>
<td>1271.25</td>
<td>1118.95</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>2095.02</td>
<td>771.30</td>
<td>632.94</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>1430.63</td>
<td>678.99</td>
<td>490.58</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>1137.48</td>
<td>658.42</td>
<td>423.19</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>988.32</td>
<td>643.89</td>
<td>389.89</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td>928.02</td>
<td>646.30</td>
<td>375.96</td>
</tr>
</tbody>
</table>

Table 1. Wall clock runtimes in seconds for each version of FIXEDGRID executing on the Cell Broadband Engine.

5.1.1. Version 1: Translate Fortran 90 to C

The current version of the Cell Broadband Engine SDK (CBE SDK 2.2) does not include a Fortran compiler. (CBE SDK 3.0 is expected to include this feature.) Without a Fortran compiler available, we were forced to translate the FIXEDGRID Fortran 90 source code to a language supported by the CBE SDK. To avoid confounding speedup attributable to language features with speedup from program optimization, we only report metrics from the C code in all the experiments.

Although there are several tools for automatically translating Fortran 90 to C, such tools often produce code that is difficult to read and maintain. We chose to translate the Fortran source by hand to better fine tune our implementation. The translation process, while long and tedious, was not difficult. The most frequent source of difficulty was the difference in 2D array storage between the languages (2D arrays are stored column-major in Fortran 90, but row-major in C). All Fortran parameter statements were changed to C preprocessor directives, and modules were encapsulated by C header files. The result was a generic C program suitable for any standard C compiler. The C code did not support multiple cores, vectorization, or any special features of the Cell, but it was approximately 62% larger than the original Fortran 90. This was primarily due to C’s lack of array syntax, a Fortran feature heavily used by FIXEDGRID. It took approximately 38 work mixedhours to finish the translation.

5.1.2. Version 2: Offload computational cores

The main computational core of FIXEDGRID is the discretize() function. discretize() calculates mass flux in a single row or column. We parallelized FIXEDGRID by exporting discretize() to the Cell SPE units, leaving the PPE responsible for coordinating SPE execution and file I/O. On startup, each SPE receives pointer to a fixedgrid_spe_argv_t structure (Listing 1) which contains the parameters to the discretize() function. fixedgrid_spe_argv_t is exactly 128 bytes long and aligned on a 128-byte boundary, making it highly efficient for DMA transfer. The PPU signals the SPU that the arguments have changed via the Cell mailbox registers. The SPU fetches the fixedgrid_spe_argv_t structure by DMA from main memory on receipt of the message.

The SPU has 256KB of local storage and cannot access main memory directly. Considering all variables, this translates to a rough limit of approximately 8000 double-precision variables per row or column which can be held in SPE local storage. Rows or columns exceeding this limit are processed piecewise. Columns of data are particularly problematic for the Cell, as memory addresses must be contiguous and aligned on a 16-byte boundary in order to be transfered via DMA. A straight-forward technique for copying array columns to local storage is to buffer the column data on the PPE and DMA the buffer to the SPE. This triples the number of copies required for column-wise updates and severely bottlenecks scalability. Figure 3(a) shows the performance of FIXEDGRID when this method is employed. There is a clear
defined datatype for passing arguments to the SPE program

Listing 1. User-defined datatypes for passing arguments to the SPE program

calability asymptote at three SPEs imposed by this method. Figure 3(b) shows that computation time decreases as the number of SPUs increases (“Row Discret.”), but the time spent copying memory to and from the SPE remains constant (“Array Copy”). We describe how columns of data can be efficiently copied to the SPU local storage in Section 5.1.4.

5.1.3. Version 3: Optimize

We optimized our code by double-buffering DMA transfers from main memory to SPU local storage, unrolling loops executed on the SPE, and using SIMD intrinsics. Double buffering provided the bulk of speedup compared to loop unrolling and SIMD intrinsics. Each SPE preemptively loads the next row or column into local storage while processing the current row or column by using nonblocking MFC function calls to overlap communication with computation. Double buffering alone reduced runtime by approximately 23%.

The SPU hardware assumes linear instruction flow and produces no stall penalties from sequential instruction execution. A branch instruction has the potential of disrupting the assumed sequential flow. Correctly predicted branches execute in one cycle, but a mispredicted branch incurs a penalty of approximately 18-19 cycles. The typical SPU instruction latency is between two and seven cycles, so mispredicted branches can seriously degrade program performance. We eliminated branches in the SPE code by function inlining and loop unrolling.

The SPU is a 128-bit SIMD vector processor capable of simultaneous arithmetic on two double-precision floating point values. We used the vector double data type and the vector arithmetic functions provided in the CBE SDK library of SIMD intrinsics to simplify vector programming on the SPU. These intrinsics signal the SPU compiler to vectorize certain operations, although the compiler can recognize and auto-vectorize code when instructed. We used both manual vectorization and compiler auto-vectorization to reduce runtime by a further 18%.

5.1.4. Version 4: Scalable Non-Contiguous DMA

While rows of the concentration matrix can be easily and efficiently copied to the SPU local store, elements of columns of the matrix are stored at noncontiguous addresses. Scalable random access to matrix columns can be achieved through DMA lists. A DMA list resides in the SPU local store and consists of a list of main memory addresses and transfer sizes which are stored as pairs of unsigned integers. The first of these sixty-four bits is the stall-and-notify flag. Setting this bit causes the DMA operation to suspend execution of the list and set a stall-and-notify event status for the SPU. We used the structure in Listing 2 to simplify DMA lists.

To transfer a column of $NY$ elements requires a DMA list of $NY$ entries. The same DMA list can be used both for copying to and from main memory. All DMA transfers must be made from addresses which are aligned on a 16-byte boundary, but double-precision variables are eight bytes long, so two columns must be retrieved simultaneously, even if only one column is desired. When the DMA list is processed, the two columns are copied to the same buffer in local storage with their elements interleaved. The desired column is then extracted by the SPE from the local storage. The overhead of this method is $O(NY \times (NSPU - 1))$ larger than using the PPU to align and DMA matrix columns, but scalability is greatly improved by moving the array copy loops to the SPEs. This scalable access method increases chemical transport model efficiency by an average of 30%, and significantly improves the scalability of dimension-splitting techniques on the Cell Broadband Engine.

5.2. Shared-Memory Multiprocessor

Parallelizing FIXEDGRID for a shared-memory multiprocessor was significantly easier than parallelizing for the Cell. We used OpenMP to spawn a fixed number of threads in the main loop. Each thread has normal access to main memory, so the concentration, wind field, and diffusion matrices are all readily available to all threads. Average runtimes for FIXED-
GRID on the shared-memory multiprocessor are shown in Table 2.

Determining shared variables and synchronization points were the most important design considerations in parallelizing FIXEDGRID. Initially, the concentration, wind field, and diffusion matrices were not shared among all threads. Each thread would buffer portions of these matrices, perform row and column discretization for one time step, and then write their results back to shared memory (similar to a distributed-memory algorithm). This approach reduced synchronization calls by two-thirds, but the large overhead of buffering and writing back to memory made this approach infeasible. Instead, we opted to have each thread operate directly on shared memory. Although this required more synchronization between threads, overhead and code complexity were significantly reduced. We note, however, that for very large numbers of cores, the former approach is likely to scale better.

<table>
<thead>
<tr>
<th>Threads</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>414.22</td>
</tr>
<tr>
<td>2</td>
<td>211.89</td>
</tr>
<tr>
<td>3</td>
<td>147.23</td>
</tr>
<tr>
<td>4</td>
<td>114.62</td>
</tr>
<tr>
<td>5</td>
<td>93.67</td>
</tr>
<tr>
<td>6</td>
<td>77.87</td>
</tr>
<tr>
<td>7</td>
<td>68.82</td>
</tr>
</tbody>
</table>

Table 2. Wallclock run times in seconds for FIXEDGRID on a shared-memory multiprocessor.

6. ANALYSIS AND COMPARISON

Our code achieves a maximum 702% superlinear speedup on the accelerator-based chipset, and a maximum sub-linear speedup of 565% on the shared-memory multiprocessor (see Figures 4(a) and 5(a)). The superlinear speedup exhibited by the heterogeneous Cell Broadband Engine is largely attributable to the overlap of communication and computation on the Cell. Double buffered DMA guarantees data is fetched into SPE local storage before it is needed, without interrupting the SPU’s execution. The sublinear performance of the homogeneous Intel Quad-Core Xeon is due to synchronization overhead. All threads must be synchronized at the end of each row/column discretization since the shared-memory code operates directly on the concentration matrix.

Figures 4(b) and 5(b) show the inclusive time spent in major portions of the code. “Array Copy” gives the time spent buffering matrix columns in preparation for DMA to SPU local storage. “Row Discretization” and “Column Discretization” give the total time (over all threads) spent executing discretize() on rows and columns, respectively. It is interesting to note that discretize() is called twice as many times for rows as for columns, so we expect “Row Discretization” to represent twice as much time. While this is nearly the case for OpenMP, the cost of row discretization is almost exactly half the cost of column discretization on the Cell, due to the cost of non-contiguous memory access.

Although FIXEDGRID runtimes on the Cell Broadband Engine are significantly longer than on the Quad-Core Xeon, the accelerator-based heterogeneous processor shows a much better efficiency and scalability. This suggests that accelerator-based heterogeneous processors are a more promising platform for multiphysics simulation than standard heterogeneous CPUs. The lower runtimes on the shared memory machine are attributable to hardware support for double-precision arithmetic. Kurzak and Dongarra [11] found that the LINPACK benchmark is fourteen times slower on the Cell when double-precision arithmetic is used compared...
to single-precision arithmetic. Their results and these experimental results suggest that a Cell processor with hardware support for double-precision arithmetic would out-perform a homogeneous processor of similar clock rate. The next generation Cell Broadband Engine is expected to include full hardware support for double-precision arithmetic.

7. CONCLUSIONS AND FUTURE WORK

The majority of the existing chemical transport models are written in Fortran and employ a domain decomposition based on dimension splitting, such as row/column decomposition. This presents two major challenges to developers who wish to port chemical transport models to accelerator-based multi-core CPUs such as the Cell Broadband Engine: limited access to main memory, and a lack of language support.

It is clear that a different approach is necessary if chemical transport models are to fully utilize the power of accelerator-based CPUs. Indeed, any application which requires random access to both rows and columns of a matrix will see limited performance benefits on the Cell. Problems involving n-dimensional arrays will see even more pronounced performance penalties as the number of dimensions, and therefore indirections, increases. Methods such as DMA lists can alleviate scalability issues, but at the cost of overhead, increased program complexity, and less-than-ideal performance. Future work should focus on finding matrix representations which provide efficient random access to strided memory locations. A “strided” DMA memory flow controller command could also remove these issues, but at the cost of hardware complexity.

The lack of language support is a less troubling issue. Compilers for Fortran and other languages popular in scien-
tific computing are already being ported to accelerator-based multicore platforms. IBM is expected to include a Fortran compiler in release 3.0 of the Cell Broadband Engine SDK. The Cell C and C++ compilers support auto-vectorization, and IBM is investigating methods of automatically offloading functions to the SPEs. Some important considerations for such a method include carefully calculating the memory requirements of a function to insure it does not exceed SPE local storage, scheduling and data dependencies between functions, and applicability of vector operations in the function execution.

8. BIOGRAPHY
John C. Linford is a PhD student of computer science at Virginia Tech and an NDSEG fellow. Under a 2007 CESRI fellowship. John studied clock synchronization and large-scale performance analysis software for the IBM BlueGene/L supercomputer at Forschungszentrum Jülich. John graduated in 2005 with degrees in Computer Science and Mathematics from Weber State University as the Crystal Crest Scholar of the Year, the school’s highest academic honor, and was an adjunct professor of computer science in 2005 - 2006. John’s graduate work involves high performance computing systems and advanced multiphysics models.

Adrian Sandu obtained the Diploma in Electrical Engineering - Control Systems from the Technical University Bucharest, Romania, M.S. in Computer Science and Ph.D. in Applied Mathematical and Computational Sciences from The University of Iowa. He worked as a computer programmer (ICI Bucharest), instructor (T.U. Bucharest), TA&RA (U. Iowa), postdoctoral research associate (Courant Institute of Mathematical Sciences). Between 1998-2003 he served as a faculty in the Department of Computer Science at Michigan Tech. In 2003 he joined Virginia Tech’s Department of Computer Science. Sandu’s research interests are in the area of computational science and engineering.

REFERENCES


