

The reverse-acceleration model for programming petascale hybrid systems

S. Pakin
M. Lang
D. J. Kerbyson

*Current technology trends favor hybrid architectures, typically with each node in a cluster containing both general-purpose and specialized accelerator processors. The typical model for programming such systems is host-centric: The general-purpose processor orchestrates the computation, offloading performance-critical work to the accelerator, and data are communicated only among general-purpose processors. In this paper, we propose a radically different hybrid-programming approach, which we call the **reverse-acceleration model**. In this model, the accelerators orchestrate the computation, offloading work that cannot be accelerated to the general-purpose processors. Data is communicated among accelerators, not among general-purpose processors. Our thesis is that the reverse-acceleration model simplifies porting codes to hybrid systems and facilitates performance optimization. We present a case study of a legacy neutron-transport code that we modified to use reverse acceleration and ran across the full 122,400 cores (general-purpose plus accelerator) of the Los Alamos National Laboratory Roadrunner supercomputer. Results indicate a substantial performance improvement over the unaccelerated version of the code.*

Introduction

Power considerations, limitations on pin bandwidth, and cost constrain the performance achievable by general-purpose processor cores. Consequently, a trend in the computer industry is to construct hybrid systems, computers containing a mixture of different core types—either within a socket or across sockets. Some cores target general-purpose workloads, which are characterized by complex control flow, large working-set sizes, irregular memory access patterns, relatively light floating-point activity, and limited parallelism. Other cores target more specialized workloads, typically those that exhibit relatively infrequent branching, small working-set sizes, regular memory access patterns, heavy floating-point activity, and abundant parallelism. By running different parts of an application on different processor cores, a hybrid system can exploit the distinct advantages of each type of core.

The primary challenge imposed by hybrid systems is programmability—how best to assign work to processor cores while taking into consideration the strengths and weaknesses of each core type and the time needed to

move data among distinct memory regions. This challenge is exacerbated when the hybrid system is organized as a cluster comprising many types of cores, many memory regions, and many data-transfer mechanisms. The Roadrunner supercomputer [1], built by IBM for Los Alamos National Laboratory, is the quintessential example of a large, hybrid system of this nature.

In this paper, we address the issue of programmability by introducing a new approach to programming hybrid systems. Rather than treating a hybrid system as a cluster of communicating general-purpose cores with attached special-purpose cores, our approach, the *reverse-acceleration model*, treats a hybrid system as a cluster of communicating special-purpose cores with attached general-purpose cores. The advantages of this contrarian world view are that it simplifies porting codes from non-hybrid systems to hybrid systems and helps improve performance by more naturally keeping the bulk of the computation for an application on the high-speed, special-purpose cores. To support these claims, we implemented a library that enables a programmer to

©Copyright 2009 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied by any means or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/09/\$5.00 © 2009 IBM

program to the reverse-acceleration model; we ported a non-hybrid application to this library, and we ran the resulting hybrid application on the Roadrunner supercomputer at Los Alamos National Laboratory. Performance results and implications are described later in this paper.

Related work

The use of additional processing capabilities or accelerators that supplement a main processor is not new. Many recent investigations have analyzed the benefits of using graphics processing units (GPUs) for high-performance, general-purpose processing; field-programmable gate arrays (FPGAs) to accelerate specific operations; and single-instruction, multiple-data (SIMD) processors to manipulate arrays and vectors at high speed. To date, most work has concentrated on optimizing small kernels on systems comprising only a few nodes. A few notable exceptions include the use of hybrid systems to improve the performance of the production Weather Prediction and Forecasting (WRF) code [2] from the National Center for Atmospheric Research (NCAR) and the Nanoscale Molecular Dynamics (NAMD) program [3] from the University of Illinois at Urbana-Champaign (UIUC). In the former study, the authors show that an order-of-magnitude gain in performance can be obtained when accelerating the WRF main computational kernel on a single node containing an NVIDIA 8800 GTX GPU. This improvement yields a 23% performance gain for the application as a whole. In the latter study, the authors explored the performance of NAMD on a system containing 64 NVIDIA GPUs. As in the WRF study, only part of NAMD was accelerated on the GPUs, but this part represents a large fraction of the total run time. Consequently, the use of GPU accelerators led to a more than 5X performance improvement over using just the host CPUs.

More recently, the Roadrunner supercomputer [1] has enabled scientists to experiment with hybrid systems at significantly large scales. By exploiting the Roadrunner Cell Broadband Engine*** (Cell/B.E.) processors [4] as computational accelerators, the vector particle-in-cell (VPIC) code achieved a total processing rate of 0.374 Pflops [5] (using single-precision operations); the scalable parallel short-range molecular dynamics (SPaSM) code achieved 0.369 Pflops [6] (double precision); and the PetaVision visual-cortex simulator achieved more than 1 Pflops (single precision), demonstrating the potential of hybrid computing.

All of the aforementioned hybrid applications are programmed using the traditional approach of having the general-purpose cores manage the computation and offloading performance-critical sections to non-

communicating accelerators. The closest approach to our reverse-acceleration model is that used by the Cell/B.E. versions of MPI (Message Passing Interface) developed by Kumar et al. [7] and Krishna et al. [8]. That programming model, like ours, assigns the main computation work to the special-purpose cores. However, it does not use the general-purpose cores at all. In contrast, we see the general-purpose cores as playing an important role in a hybrid computation, albeit a subservient one to that of the special-purpose cores. Another similar approach to ours is taken by Ohara et al. [9] in their MPI microtask work. In the MPI microtask model, the general-purpose core schedules work on the special-purpose cores, but the special-purpose cores can communicate with each other and run the bulk of the computation. Kahle et al. [4] refer to this approach as the *computational acceleration model*. A key problem with the computational acceleration model is that a centralized scheduler presents a performance bottleneck that limits scalability. In contrast, in the reverse-acceleration model, the accelerators drive the computation, so there is no need for a centralized scheduler. Programs written with the reverse-acceleration model can, therefore, scale gracefully, which we demonstrate in the section "Evaluation," later in this paper.

The difficulty of exploiting accelerators and the flexibility of the Cell Broadband Engine Architecture has resulted in the development of a number of hybrid programming environments for the Cell/B.E. Cell Superscalar (CellSs) [10] and the Accelerated Library Framework (ALF) [11] both facilitate splitting programs into units of code and data destined for either the general-purpose or the special-purpose cores. CellSs uses function annotations in C programs to specify what dependencies exist in the execution. ALF takes a library approach to simplifying hybrid programming by handling common data-transfer patterns automatically. Newer versions of IBM XL compilers for the Cell/B.E. support automatic hybridization of OpenMP** programs [12]. That is, the programmer writes a single program with parallelism specified using ordinary OpenMP directives embedded in the code [13]. The compiler and run-time system automatically designate threads to run on the special-purpose cores and handle the transfer of data between memory spaces. Hence, the same code can run both on non-hybrid systems and on the hybrid Cell/B.E. processor. Similarly, programming environments from Gedae [14] and RapidMind [15] are designed so that programs written using their tools can run on non-hybrid and on multiple hybrid systems, including the Cell/B.E. as well as GPUs and FPGAs.

While this paper describes our reverse-acceleration implementation of Sweep3D [16] on the Roadrunner

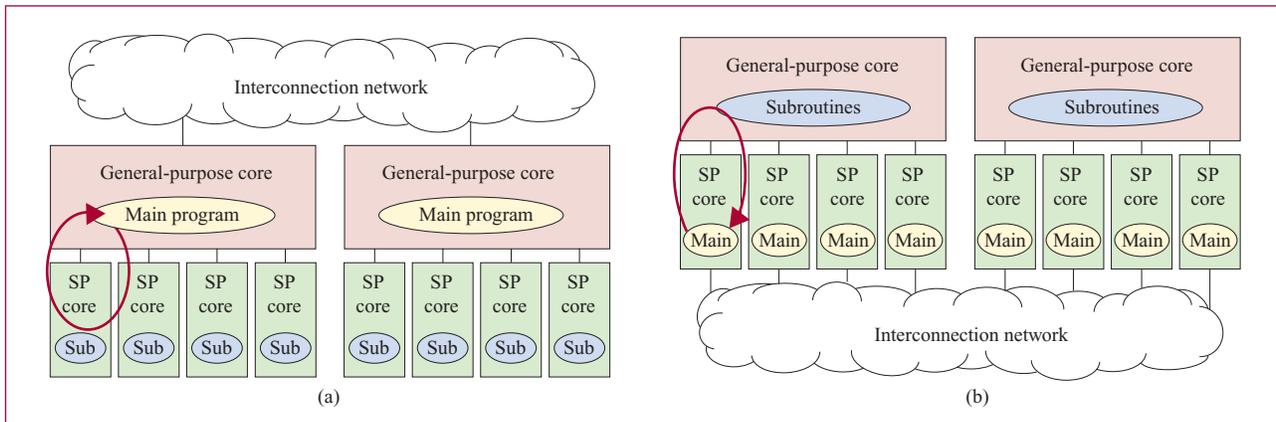


Figure 1

Abstractions provided by the (a) accelerator model and the (b) reverse-acceleration model. (Sub: subroutines; SP: special purpose.)

supercomputer, analytical performance modeling has previously been used to evaluate the performance of Sweep3D under a range of possible large-scale compute node and accelerator configurations and using the ClearSpeed** CSX600 as an example [17]. Also, a hybrid Cell/B.E. version of Sweep3D was previously implemented with the traditional approach of having the general-purpose core offload compute-intensive work to the special-purpose cores [18].

Reverse acceleration

The traditional approach to programming a hybrid system is what we term the *accelerator model*. The accelerator model treats the general-purpose cores as main processors and the special-purpose cores as accelerators whose role is to speed up pieces of the application. In the accelerator model, the programmer starts by writing code for the general-purpose cores. In a cluster environment, this would normally be parallel code in which the processes running on the general-purpose cores exchange data via explicit message passing. The programmer then identifies performance-critical routines that are likely to see an improvement in performance by running on the special-purpose cores and correspondingly recodes those routines for the accelerators.

The enticement of the accelerator model is that unmodified applications can run immediately. Performance improvements made by offloading compute-intensive routines to the accelerator can be implemented incrementally. Unfortunately, there are two problems inherent in the accelerator model. First, accelerated routines require structural changes from the original code. While the unaccelerated routine is sequential (though possibly forming a part of a larger, parallel

application) the accelerated routine is data or task parallel and may utilize explicit communication. After parallelizing the accelerated routine, the routine then needs to be augmented to distribute data to the special-purpose cores and aggregate the results. Second, the process of refining performance-critical routines may result in wasted effort. That is, after the programmer has rewritten a routine for an accelerator, it may turn out that the performance gained from the improved computational performance may be dominated by the cost of moving data between the general-purpose core and the special-purpose core.

To address the problems of restructuring code for accelerators and the difficulty of determining suitable routines for acceleration, we propose turning the accelerator model upside-down. Instead of treating a hybrid system as a cluster of communicating general-purpose cores, each with an attached accelerator for offloading sequential, compute-intensive work, one can treat a hybrid system as a cluster of communicating, high-speed, special-purpose cores, each with an attached general-purpose core for offloading control, memory, or I/O (input/output) intensive work. We call this approach the *reverse-acceleration model*.

Figure 1 illustrates how a programmer views the system in the accelerator model and in our reverse-acceleration model. In both cases, the hardware is the same; what changes is the abstraction presented to the programmer. In the accelerator model [Figure 1(a)], the general-purpose cores manage the computation, sending compute-intensive work to the special-purpose cores (labeled “SP” in the figure) and aggregating the results. In the reverse-acceleration model [Figure 1(b)], the special-purpose cores manage the computation, sending control-

intensive work to the general-purpose cores and aggregating the results. In the accelerator model, general-purpose cores communicate with other general-purpose cores, while the special-purpose cores communicate only with their associated general-purpose cores. In the reverse-acceleration model, special-purpose cores communicate with other special-purpose cores, while the general-purpose cores communicate only with their associated special-purpose cores.

Note that Figure 1 presents only logical views of a hybrid system. In practice, the underlying hardware need not exactly mimic either Figure 1(a) or Figure 1(b). For instance, there may be any number of special-purpose cores associated with each general-purpose core; there may be any number of nodes in the system; there may be any number of processor sockets per node; and any number of cores may be packaged onto a single socket. In some implementations, general-purpose and special-purpose cores may both be packaged onto the same socket. Broadly speaking, though, the accelerator model presents the programmer with a lower-level abstraction (i.e., closer to the underlying hardware), while the reverse-acceleration model presents the programmer with a higher-level abstraction (i.e., closer to a simpler, non-hybrid system).

There are two advantages to the reverse-acceleration model: programmability and ease of performance optimization. The reverse-acceleration model aids programmability by enabling non-hybrid, parallel codes to run immediately on the high-speed special-purpose cores while delaying performance optimization until afterwards. The programming approach that we have found to work well proceeds as follows:

1. Get the program minimally running on the special-purpose cores. This may involve scaling down the problem size to fit in on-chip memory and enabling code overlays so that larger codes can run. The goal in this step is for the program to reach a baseline state as rapidly as possible.
2. Generalize the data motion of the program to match the capabilities and limitations of the special-purpose cores. This may involve, for example, performing explicit bulk data transfers between on-chip and off-chip memory. The goal in this step is for the program to run with full-sized inputs.
3. Optimize computation in the program to take advantage of the capabilities and limitations of the special-purpose cores. This may involve, for example, vectorizing loops or employing heavy multithreading. The goal in this step is for the program to run as fast as possible.

The second advantage of the reverse-acceleration model is that it facilitates performance optimization. Under the accelerator model, a programmer must identify functions that may be able to exploit the special-purpose cores, port those functions to the special-purpose cores, modify the code to transfer program data to the special-purpose cores and the results back from the special-purpose cores, and finally measure the performance and determine whether offloading those functions to the special-purpose cores was in fact beneficial, reverting the code to its prior state if not. In a sense, the accelerator model *discourages* function offloads because only functions that are very likely to see a performance improvement are worth the additional coding effort needed to manage the extra data transfers.

The reverse-acceleration model, in contrast, *encourages* running code on the special-purpose processors. Because all of the program's code and data are already on the special-purpose cores before the performance-optimization process begins, no additional programming overhead is needed to ensure that functions running on the special-purpose cores can access their data. Performance optimization follows the more traditional approach of profiling an execution, determining which functions took the most time and optimizing the performance of those functions. There is no need to guess which functions are likely to run fast or slow, as actual measurement data are available.

Consider also the role of idle time in a computation on a hybrid system. In the accelerator model, the (fast) special-purpose cores lie idle until a general-purpose core offloads work to them. In the reverse-acceleration model, it is instead the (slow) general-purpose cores that lie idle waiting for work assignments from the special-purpose cores. This is another point in favor of the reverse-acceleration model: It is better to keep the cores with higher compute rates busy than those with lower compute rates.

A downside of the reverse-acceleration model is that it imposes some specific hardware requirements that are not met by all available special-purpose cores. It requires that the special-purpose cores be capable of running independent threads of control, that they have direct access to some amount of private memory (albeit not necessarily as large as what the general-purpose cores have access to), and that they have some mechanism for bidirectional communication with a general-purpose core. Preferably, they also have the ability to communicate without the involvement of a general-purpose core and some I/O capability, but these features are not strictly required. Many current and soon-to-be-released special-purpose processors already support these requirements. While this paper focuses on the Cell/B.E. processor [4], other processors such as the Intel Larrabee [19] and the

Tilera TILE64** [20] should also be able to implement the reverse-acceleration model in a straightforward manner.

Implementation

To demonstrate the viability of the reverse-acceleration model, we implemented a messaging library that presents the programmer with the abstraction that the special-purpose cores can communicate directly with each other across an entire cluster.

Roadrunner

The testing ground for our reverse-acceleration model is the Los Alamos National Laboratory Roadrunner supercomputer [1], which at the time of this writing is not only the world's largest hybrid system but also the world's fastest system of any type, according to the TOP500** list of supercomputers [21].

The combination of flexible general-purpose (AMD Opteron** [22]) and high-performing special-purpose (IBM PowerXCell* 8i [23]) processors is the foundation of the Roadrunner system. The goals of the design were to provide high computational performance within acceptable cost and power budgets, and the use of hybrid processor technology was found to be a suitable approach to meet those constraints.

Although Roadrunner contains an equal number of conventional general-purpose microprocessor cores and special-purpose accelerators, the vast majority of the available performance results from the special-purpose accelerators, the PowerXCell 8i processors. These provide more than 95% of the peak performance and more than 85% of the peak memory bandwidth. The entire system has a peak performance of 1.38 Pflops (double precision; 2.91 Pflops single precision). In addition, the high processing efficiency that is possible for many applications results in immense achievable performance. In May 2008, Roadrunner was the first system to achieve over 1 Pflops sustained performance on the industry-standard Linpack benchmark [24].

The full system consists of 3,060 compute nodes that are arranged into 17 compute units (CUs). The 180 nodes within each CU are interconnected in a full fat-tree topology [25] using a single 288-port InfiniBand** 4X double-data-rate (DDR) switch [26]. CUs are interconnected using a further eight switches organized as a 2:1 reduced fat tree. The switch fabric can support up to 24 CUs without alteration, although the currently installed system contains only 17, as shown in **Figure 2**. Each InfiniBand switch contains 36 crossbar chips. The intra-CU switches are arranged into two levels of crossbars (one containing 24 and one containing 12), and the inter-CU switches are arranged into three levels of 12 crossbars.

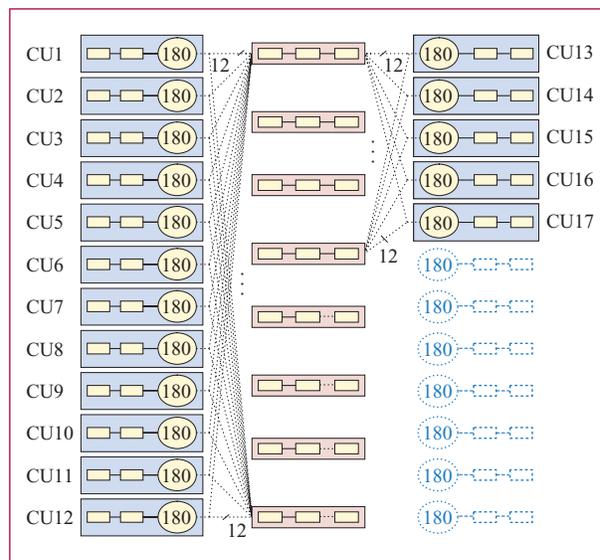


Figure 2

Configuration of Roadrunner 17 compute units (CUs) interconnected with InfiniBand.

A Roadrunner compute node, known as a *triblade*, consists of three blades, as shown in **Figure 3**. One blade, an IBM LS21 [27], contains two dual-core AMD Opteron processors, and the other two blades, both IBM QS22s [28], each contain two PowerXCell 8i processors [23]. The PowerXCell 8i sees a 7X improvement in double-precision floating-point performance over the original Cell/B.E. processor (used in the Sony Playstation*** 3 [29]), which has been extensively analyzed for scientific computation, for example, by Williams et al. [30, 31]. An expansion card, taking the space of a fourth blade, serves to interconnect the three compute blades as well as connect to other triblades through an InfiniBand host channel adapter (HCA) [26]. The peak performance of a node is 449.6 Gflops (double precision).

The amount of memory on the Opteron blade (16 GB) equals the total amount of memory on both the PowerXCell 8i blades (8 GB apiece). The Opteron processors are clocked at 1.8 GHz with each core able to issue two double-precision floating-point operations per cycle, resulting in a peak of 14.4 Gflops per LS21 blade. Each core has a 64-KB L1 data cache, a 64-KB L1 instruction cache, and a 2-MB L2 cache. The PowerXCell 8i processors are clocked at 3.2 GHz and contain one IBM PowerPC* processor element (PPE) and eight synergistic processor elements (SPEs). The PPE has a traditional cache-based memory hierarchy consisting of a 32-KB L1 data cache, a 32-KB L1 instruction cache, and a 512-KB L2 cache. It can issue two double-precision floating-point operations per cycle.

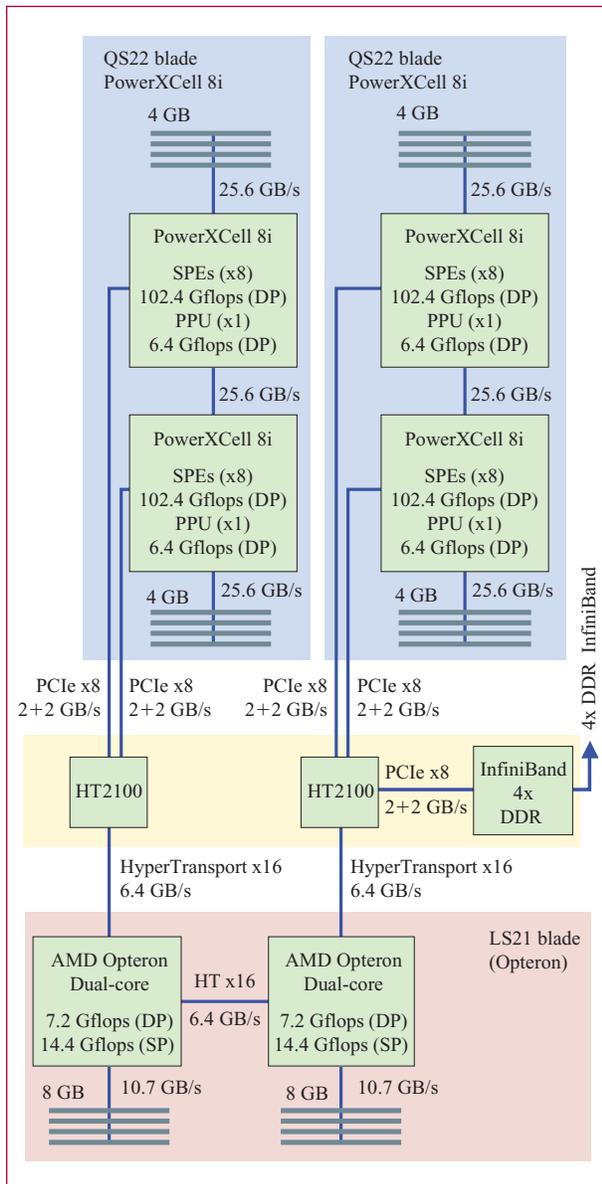


Figure 3

Illustration of a Roadrunner compute node (triblade). (DP: double precision; SPE: signal processing element; PPE: IBM PowerPC processor element; PPU: PowerPC processor unit; DDR: double data rate.)

Each SPE contains a SIMD processing unit that can issue a total of four double-precision or eight single-precision floating-point operations per cycle. Thus, the peak performance per the PowerXCell 8i is 108.8 double-precision Gflops, of which 102.4 Gflops are from the eight SPEs. A corollary is that the PPE core is too underpowered to use for serious computation; it best serves as a controller for the SPEs. A key characteristic of

the SPE is that it can directly address only 256 KB of memory. This high-speed memory, known as *local store*, takes the place of a conventional cache architecture. Main memory, shared with the PPE, can be accessed only via explicit direct memory access (DMA) transfers to or from local store.

Roadrunner has a deep communication hierarchy. Within a PowerXCell 8i, the SPEs, PPE, and other logic are connected via an arbitrated bus known as the *element interconnect bus* (EIB). The EIB contains four rings (two running clockwise and two counterclockwise) and supports an aggregate peak bandwidth of 204.8 GB/s, although a single transfer cannot exceed 25.6 GB/s [32]. The two PowerXCell 8i sockets on a QS22 blade are directly connected via a FlexIO interface [33], which, as configured in the Roadrunner system, provides an aggregate peak bandwidth of 25 GB/s, with single transfers limited to 6.25 GB/s. Within a triblade, each of the QS22 blades is connected to the LS21 blade via two PCI Express** (PCIe**) x8 connections [34], as shown in **Figure 3**. The PCIe buses from the Cell/B.E. blades are converted to HyperTransport** for connection to the Opteron processors using two Broadcom** HT2100 I/O controllers. The HT2100 has a single HyperTransport x16 port and three PCIe x8 ports. Each QS22 blade has a direct connection to an Opteron socket on the LS21, providing a peak bandwidth between each PowerXCell 8i processor and its associated Opteron core of 2 GB/s in each direction. The third port on one of the HT2100 controllers connects a Mellanox** 4x DDR InfiniBand HCA. Connectivity between triblades, therefore, exhibits a peak bandwidth of 2 GB/s in each direction.

Cell Messaging Layer

To provide the illusion that Roadrunner is a “sea of SPEs” and enable it to be programmed as such, we developed a software messaging layer called the *Cell Messaging Layer* (CML) [35] that enables Roadrunner to be programmed using the reverse-acceleration model. CML additionally runs on homogeneous Cell/B.E. clusters (i.e., those containing only Cell/B.E. processors without associated general-purpose processors) including, for example, clusters of IBM QS21 Cell/B.E. blades [36] interconnected via InfiniBand [26] and clusters of Sony Playstation 3 gaming consoles [29]. CML is freely available for download from SourceForge (<http://cellmessaging.sourceforge.net/>).

CML implements a subset of the MPI messaging layer [37] as an SPE library. MPI is the de facto standard for programming parallel computers and workstation clusters. Implementing an SPE version of MPI, therefore, introduces hybrid programming to a large number of scientific application developers, who can now focus their efforts on Cell/B.E.-specific optimizations rather than on

implementing basic communication and synchronization mechanisms.

The implementation goal is initially to provide the most useful MPI functions and incrementally introduce additional functions as needed. The functions currently implemented are `MPI_Abort()`, `MPI_Allreduce()`, `MPI_Barrier()`, `MPI_Bcast()`, `MPI_Comm_get_attr()`, `MPI_Comm_rank()`, `MPI_Comm_size()`, `MPI_Finalize()`, `MPI_Init()`, `MPI_Recv()`, `MPI_Reduce()`, `MPI_Send()`, `MPI_Wtime()`, and `MPI_Wtick()`. The PMPI profiling interface is supported for all of those functions. However, the MPI wildcard receive, `MPI_ANY_SOURCE`, is not supported, and only a limited number of values can be used as message tags (a configuration-time option). `MPI_Send()` provides no buffering, so it is identical to the `MPI_Ssend()` function.

Although the MPI interface defines hundreds of functions, it is not an issue to include all of them in an SPE library, even considering that an SPE has access to only 256 KB of on-chip memory. The key is that only functions that are actually invoked take up any space in local store. CML is designed so that each object file in the library provides only a small number of functions. At program link time, the linker ignores unreferenced objects. Because most MPI programs utilize only a tiny fraction of the functions that MPI provides, CML needs to reserve only a small amount of local store for itself. For example, initialization, finalization, and the two point-to-point communication functions together take up only 5 KB (2%) of the local store. Programs with severe memory-capacity constraints can sacrifice communication performance for available local store and demand-load CML functions using code and data overlays.

To further support the reverse-acceleration model, CML provides a few features that are not part of MPI. The most important of these is support for a Remote Procedure Call (RPC) mechanism for invoking functions on the general-purpose processor and receiving the results. In fact, on Roadrunner, CML not only enables SPEs to invoke functions on their associated PPE but also allows PPEs to invoke functions on their associated Opteron processor. These RPC invocations can be chained together so an SPE can invoke a function on an Opteron processor indirectly via a PPE function. Some examples of how we have used the CML RPC mechanism include having SPEs call the PPE `malloc()` function to allocate main memory and receive a pointer to the allocated memory and having SPEs call I/O functions on the Opteron processors to manage the I/O files of a program.

The CML implementation is designed to run extremely fast. It takes advantage of a technique called *receiver-initiated message passing* [35] to reduce the number of

internal synchronizations needed to implement message passing semantics atop RDMA hardware. Essentially, the receiver transmits a message request to the sender, who transfers its data directly into the receiver-specified buffer. SPE-to-SPE communication within a Cell/B.E. socket or between FlexIO-connected Cell/B.E. processors on a single blade proceeds with absolutely no PPE involvement. This implies not only that serialization at the PPE interface is eliminated but also that the full bandwidth of the EIB is available to concurrently communicating pairs of SPEs. In contrast, programs using the accelerator model tend to push work from the PPE to each underlying SPE; each SPE computes independently; and the results are sent from all SPEs over a single interface back to the PPE. The EIB is underutilized in this usage model. By facilitating intra-socket and intra-blade communication, the reverse-acceleration model encourages exploiting the high-speed short-range networks.

Extreme scalability is another feature of the CML implementation. CML follows an Internet gateway-style approach to enable large numbers of peers to communicate without per-process memory demands growing with system size. That is, an SPE needs to know how to route messages only to its associated PPE, a PPE needs to know how to route messages only to its associated SPEs and Opteron processor, and an Opteron processor needs to know how to route messages only to other Opteron processors and its associated PPE. Each SPE holds an array of communication states that enable it to communicate with all other local SPEs (i.e., those on the same Cell/B.E. or same blade) plus one additional array element for communicating with all non-local SPEs. Each PPE, which is a relatively slow core, is responsible merely for forwarding messages between its associated SPEs and Opteron processor over the PCIe bus using the IBM Data Communication and Synchronization (DaCS) library. Finally, each Opteron processor uses MPI to forward messages to the remote Opteron processor associated with the target SPE.

The following is a more detailed technical description of the CML internal structure. There are two main cases for point-to-point communication: Either the sending and receiving SPEs reside within the same address space (a socket or a blade) or they reside within different address spaces. In the first case, upon executing an `MPI_Recv()`, the receiver puts a message descriptor (the target address, message length, send/receive completion flags, and other such data: 64 bytes total) into a per-SPE descriptor array in the sender's local store. The receiver then spins waiting for the send-completion flag to be set. Upon executing an `MPI_Send()`, the sender reads the message descriptor, puts the message data to the target address, and performs a fenced `Put` of the send-completion flag.

(Fencing imposes order between sets of RDMA transfers.) The receiver sees that the send-completion flag has been written and, therefore, exits its `MPI_Recv()`.

The second case, communication between SPEs in different address spaces (separate blades), is similar but is complicated by the fact that multiple SPEs feed into a single PPE. (Only the PPE connects to the outside world.) Upon executing an `MPI_Recv()`, the receiver first has to wait for a free slot to become available in the send queue of its PPE. The receiver then puts a message descriptor into that slot and starts spinning on the associated completion flag. The sender follows analogous steps. Upon executing an `MPI_Send()`, it first has to wait for a free slot to become available in the send queue of its PPE after which it puts the data into that slot. In Roadrunner, the PPE uses the DaCS asynchronous send/receive calls to transfer the message to its associated Opteron processor. The Opteron processor uses the MPI asynchronous `MPI_Isend()` and `MPI_Irecv()` calls to transfer the data to the receive-side Opteron processor, via either shared memory or the InfiniBand network, depending on whether the communication is intra-node or inter-node. The receive-side Opteron processor uses the DaCS asynchronous send/receive calls to transfer the message to the target PPE. The PPE uses one of two mechanisms to transfer the message to the receiving SPE: programmed I/O [i.e., `memcpy()`] for small messages (≤ 128 bytes) and an SPE-initiated RDMA Get for large messages.

A prior publication presents further details on the data structures and communication mechanisms used by CML for both point-to-point and collective communication [35].

Evaluation

It is difficult to quantify precisely the productivity improvements gained by programming hybrid systems using the reverse-acceleration model instead of the more traditional accelerator model. We present the following as anecdotal evidence that the reverse-acceleration model is easy to use. A first-year Ph.D. student managed to port both a molecular dynamics code [38] and a lattice Boltzmann code [39] to the Roadrunner architecture using CML in approximately one day apiece as part of an effort to introduce a performance-profiling mechanism to CML-based applications [40]. That is, he quickly completed Step 1 of the methodology presented in the section on reverse acceleration, leaving additional time to work on the CML-independent Steps 2 and 3. Sweep3D, which we describe in detail below, was ported to the Cell/B.E. concurrently with the development of CML, which renders meaningless any statement of the time involvement. We can say, however, that the port was straightforward, followed the methodology presented in

the section on reverse acceleration, and required no rethinking of the basic computational structure of Sweep3D.

The focus of our evaluation is not on the productivity gains achievable by using the reverse-acceleration model but rather on quantifying the performance of our implementation of the reverse-acceleration model, CML, and demonstrating that applications written to the reverse-acceleration model can observe good performance even on extreme-scale hybrid systems. In this section, we describe an MPI application that we ported to Roadrunner and present the performance of both that application and the underlying software (CML).

Sweep3D

We focus on the Sweep3D application kernel [16] to motivate the previously described application model. Sweep3D is a deterministic implementation of discrete-ordinates neutron transport. (See Hoisie et al. [41] for a detailed description of the algorithm and thorough analysis of Sweep3D performance characteristics.) The major field to be calculated is the particle flux at spatial point (x, y, z) with energy E traveling in direction Ω . The numerical solution of the transport equation involves an iterative procedure that has been referred to as the *source iteration*. The vast majority of the time is spent in a “sweep” computation, which involves calculations iterating through the spatial coordinates for each discrete angle. This class of problems is referred to as *wavefront algorithms* due to the progression of the calculation as a wave over the spatial domain. Although Sweep3D utilizes a 3D global domain ($I \times J \times K$), the program is parallelized using a 2D domain decomposition. Two spatial dimensions (I and J) are partitioned into subdomains across a P_x by P_y processor grid. The third dimension (K) is processed in blocks for better pipelining of the computation. The basic problem is broken down spatially into data cells. At a given angle, the flux for each data cell is calculated from the flux at each of six data-cell faces and the center. This calculation is dependent on the incoming flux values from other adjacent data cells. These dependencies constrain the amount of parallelism that can be exploited. The available parallelism is easily seen in a dependency diagram (**Figure 4**). In Figure 4, the black data cells have finished calculating because they have no incoming dependencies; the red data cells are waiting for flux inputs; the blue data cells have yet to start processing this angle. Arrows indicate the dataflow: A data cell cannot calculate its flux value until all of its upstream neighbors have calculated their flux values. The maximum available parallelism for a single wavefront in the configuration illustrated in Figure 4 is four, achieved when all data cells on the major diagonal are processing angle Ω .

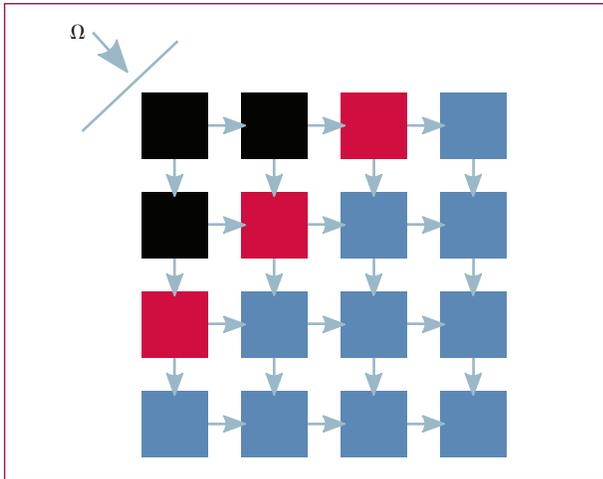


Figure 4

Dataflow in a wavefront algorithm.

The Sweep3D algorithm does not map easily to a hybrid system. When programming using the accelerator model, performance hotspots of an application are broken off and sent to the accelerator. The challenge is in propagating the flux values, which requires interprocess communication. An early, accelerator-model version of Sweep3D was shown to achieve improved performance on a single Cell/B.E. processor [18]. In that work, the PPE is used as a task controller and the work is decomposed into single I lines plus their dependencies and sent down to the SPEs for processing. The inner I loop of the $I/J/K$ loop is accelerated as the hotspot, and the results of that loop are returned to the PPE. Double buffering is used to mitigate the cost of the transfers between the SPEs and main memory. This method works for a single socket but delegates a large amount of work to the PPE, which becomes the performance bottleneck [18].

In contrast to that earlier effort to produce a Cell/B.E. version of Sweep3D, our implementation uses the reverse-

acceleration model and, thereby, manages to retain the data decomposition used in the original, non-hybrid Sweep3D. While the data decomposition of the original program is known to yield good performance on non-hybrid parallel systems, it is tricky to use the same data decomposition when programming using the accelerator model. CML and the reverse-acceleration model facilitate using a known-good data decomposition for Sweep3D, and this is the key to the performance and scalability of our implementation.

In our implementation, each SPE has a unique MPI rank and can communicate via CML to all other SPEs not only within a socket but across all of the 97,920 SPEs of the Roadrunner. With all of the Sweep3D code running on the SPEs, the algorithm is constrained only by the algorithmic dependencies and the communication costs. As with the accelerator-model version of Sweep3D, our version was optimized for the Cell/B.E. architecture [42], taking advantage of the SPE SIMD instructions and tuning for dual issue.

Table 1 summarizes the qualitative differences between the accelerator version of Sweep3D and our reverse-acceleration version.

Ensuring that the key data structures of Sweep3D fit in the SPE local store was a challenge. It is not possible to fit a subgrid much larger than $9 \times 9 \times 9$ in the local store, so in Step 2 of the methodology presented in the section on reverse acceleration, we had to modify the code to stage pieces of the subgrid from main memory to the local store. The resulting SPE executable's footprint in the local store totals 55.0 KB, which represents 49.8 KB of code (text), 0.8 KB of initialized data (data), and 4.4 KB of uninitialized data (text, data, and bss). These numbers include both Sweep3D proper and CML. The remaining 201 KB of local store is allocated as a staging area for Sweep3D subgrids.

Performance

We present performance in a bottom-up fashion. Specifically, we start by stating the theoretical peak

Table 1 Key differences between the two hybrid versions of Sweep3D.

Attribute	Accelerator version [18]	Reverse-acceleration version [43]
Code running on SPEs	Inner I loop	Entire program
Communication type	No inter-SPE communication	Intra-socket, inter-socket, and cross-cluster SPE communication
Data movement	Volume (I line and dependencies) moved twice	Surfaces moved directly to the SPEs that require them
PPE involvement	Controls SPE workers; manages data	Minimal involvement and only internal to CML message-passing library
Scale explored (cores)	Single socket (1 PPE + 8 SPEs)	Full Roadrunner (12,240 Opteron cores + 12,240 PPEs + 97,920 SPEs)

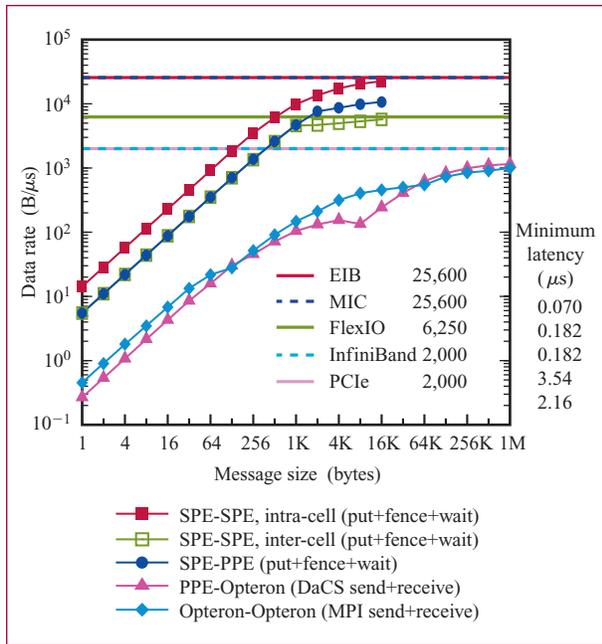


Figure 5
Performance of the various lower-level communication layers in the Roadrunner.

performance of the hardware. Next, we show performance measurements of the lowest-level software communication layers, and then we present the CML performance. After completing the presentation of communication performance, we contrast the single-socket performance with the accelerator and reverse-acceleration versions of Sweep3D that were described in the previous section. Finally, we provide Sweep3D scaling data to demonstrate how well the reverse-acceleration model can perform at an extreme scale.

Primitive performance

Figure 5 plots the data rates achievable across each core boundary. Horizontal lines indicate the theoretical peak data rate across each type of interconnect in the Roadrunner system: the EIB [32] for intra-cell SPE-to-SPE communication (i.e., within a cell socket), FlexIO [33] for inter-cell SPE-to-SPE communication (i.e., between cell sockets on the same blade), the memory interface controller (MIC) [32] for SPE-to-PPE communication, the PCIe bus for PPE-to-Opteron communication [34], and the InfiniBand network for Opteron-to-Opteron communication [26]. Points on curves represent measured data. Memory flow controller (MFC) DMA commands [43] are used for SPE-to-SPE and SPE-to-main memory data transfers. These commands provide a `Put/Get` interface and were,

therefore, measured as the steady-state time to perform a fenced `Put` [`mfc_putf()`] and wait for it to complete locally [`mfc_read_tag_status_all()`]. The IBM DaCS library is used to transfer data between a PPE and its associated Opteron processor [44]. Although DaCS supports both `Put/Get` [`dacs_put()/dacs_get()`] and `send/receive` [`dacs_send()/dacs_recv()`] interfaces, we measured only the latter because the implicit flow control and buffer management are particularly useful for constructing a higher-level communication interface such as that used by CML. Performance was measured as half the roundtrip time to transfer a message, including waiting for local completion with `dacs_wait()`. Finally, MPI—specifically the Open MPI implementation [45]—is used for Opteron-to-Opteron communication, both within and across nodes. We measured this performance as half the roundtrip time to transfer a message between two nodes using `MPI_Send()` and `MPI_Recv()`. Figure 5 also tabulates, on the right-hand side, measurements of the time needed to transmit a minimal-sized message across each of the different interconnects.

The data in Figure 5 indicate that the per-link performance bottleneck is the rate at which data can be transferred between the PPE and the Opteron processor, especially at message sizes between 2 and 16 KB. Also, all of the SPE-initiated `Put` operations take relatively little time to initiate—less than 100 cycles at 3.2 GHz—but because these are `Put` operations, not `send` operations, they do not include the time needed for flow control, buffer management, or receiver notification and, therefore, underestimate communication time in the context of application execution.

CML internally uses the MFC DMA commands, DaCS, and MPI to communicate across the various Roadrunner interconnects but presents the application programmer with true SPE-to-SPE messaging semantics. That is, one SPE can send a message using `MPI_Send()`, and another SPE—located anywhere else in the system—can receive the message with `MPI_Recv()`. CML takes care of using the appropriate transfer mechanisms (MFC DMA, DaCS, or MPI) based on the relative locations of the SPEs: the same Cell/B.E. processor, different Cell/B.E. processors on the same QS22 blade, Cell/B.E. processors on different QS22 blades in the same node, or Cell/B.E. processors in different nodes. CML implements buffer management to ensure that multiple messages sent to the same receiver do not overwrite each other. It provides flow control to stall a sender if the receiver lags behind and runs out of places to store incoming messages. Finally, per the MPI semantics [37], it supports selective message reception based on the sender's rank and a program-specified message tag. We now examine the performance implications of providing those ease-of-programming features.

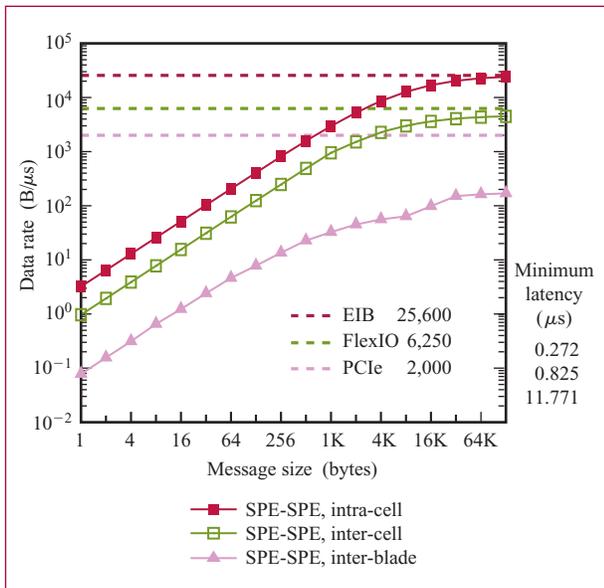


Figure 6

Performance of the Cell Messaging Layer. (EIB: element inter-connect bus.)

Figure 6, which can be contrasted with **Figure 5**, presents the data rates measured between different pairs of SPEs while performing roundtrip message transfers using CML. Note however that the *x*-axis in **Figure 6** has less range than that in **Figure 5**. This is because we measured only power-of-2 message sizes, and a 128-KB message is the largest that fits alongside the benchmark program in the 256 KB of local store in an SPE.

Both within a Cell/B.E. chip and between Cell/B.E. chips on the same blade, the maximum data rate achieved is close to the theoretical maximum for the hardware. However, inter-blade communication is noticeably slower than the bottleneck rate, that of the PCIe bus. The primary reason for this is that DaCS and Open MPI both have high startup costs for message transfers, as indicated by the minimum-latency numbers shown on the right side of **Figure 5**. Because an inter-blade message transfer incurs two DaCS latencies (PPE-to-Opteron on the send side and Opteron-to-PPE on the receive side) plus one Open MPI latency (Opteron-to-Opteron), this accounts for 9.24 μ s of the 11.77 μ s reported. The two SPE-PPE synchronizations (send and receive side) account for most of the remainder.

Because the startup costs are so high, the number of DaCS and Open MPI transfers performed per CML message needs to be kept to a minimum. Consequently, CML cannot feasibly overlap the PPE-to-Opteron, Opteron-to-Opteron, and Opteron-to-PPE transfers. This

lack of overlap is the source of the low inter-blade data rate observable in **Figure 6**. The implication is that the reverse-acceleration model, like the accelerator model, does not shield programmers from having to consider locality implications when programming a hybrid system. However, because the reverse-acceleration model presents an interface that resembles non-hybrid programming, a programmer can quickly port a non-hybrid program to a hybrid system in a locality-oblivious fashion and then incrementally optimize the code to make it aware of locality. To aid in this process, CML provides an additional CML-specific MPI communicator called `MPI_COMM_MEM_DOMAIN` that enables communication—in particular, collective-communication operations—to be restricted to a set of SPEs that share a memory space (either a single Cell/B.E. chip or the two Cell/B.E. chips on a QS22 blade, depending on how the program is run).

Application performance

As discussed earlier, we produced a hybrid version of the Sweep3D neutron-transport code using CML and the reverse-acceleration model, while Petrini et al. [18] produced a hybrid version of the same program using the MFC DMA commands and the accelerator model. Consequently, Sweep3D can serve as a comparison point for the performance of the reverse-acceleration model versus the accelerator model. We have to limit this comparison to a single cell, as this is all Petrini et al. had access to at the time of their study (which predates our work). To make the comparison fair, we ran with the same problem ($50 \times 50 \times 50$ data cells, 10 *k*-planes, 3 angles) that Petrini et al. used and the same version of the Cell/B.E. processor (the original implementation, not the enhanced PowerXCell 8i that is used in Roadrunner).

While the Petrini et al. accelerator-model Sweep3D performed 10 iterations in 1.3 seconds, our reverse-acceleration-model Sweep3D performed the same number of iterations in only 0.37 seconds—a 72% reduction in run time. Petrini et al. note that their accelerator-model run time is dominated by DMA speed. A total of 0.7 seconds—about twice our *total* execution time—is spent transferring data between the local store and main memory. The key to the improved performance of our implementation is that the reverse-acceleration model facilitates using high-speed data transfers among local stores in place of slower transfers between local store and main memory, which are simpler to implement when programming to the accelerator model.

As a final test of the reverse-acceleration model, we examine how well our version of Sweep3D scales and how its performance compares to that of the original, non-hybrid version of the program. Scalability is a key challenge for Sweep3D on any system: It communicates

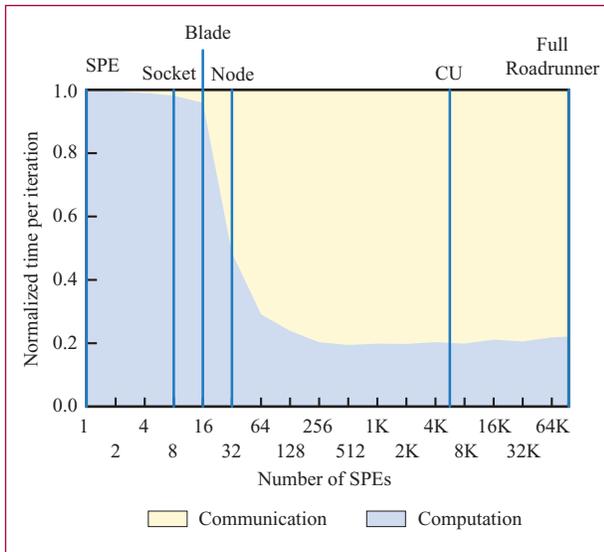


Figure 8

Contributions of communication and computation to Sweep3D run time.

acceleration model does scale, even when used to run an application that produces a large volume of messages and that exhibits limitations on parallelism. Another way to interpret the 20% efficiency number is that a hypothetical supercomputer containing 97,920 SPEs interconnected directly with an infinitely fast network (zero latency and infinite bandwidth) would run Sweep3D only five times as fast as the actual Roadrunner system runs it with CML.

Conclusions

As hybrid systems gain popularity, the issue of programmability becomes increasingly important. A natural way to program a hybrid system is to have a general-purpose processor core manage the computation, offloading compute-intensive work to a special-purpose core. We call this approach the accelerator model because it treats a special-purpose core as an accelerator for pieces of the computation that would normally run on a general-purpose core. Unfortunately, the accelerator model exhibits two shortcomings when viewed from the perspective of porting an existing parallel but non-hybrid code to a hybrid system. The first shortcoming is that the hybrid code will normally require a different parallel structure from the non-hybrid version. That is, accelerating a sequential routine typically requires parallelizing it independently of how the overall program is parallelized. The second shortcoming of the accelerator model is that the primary data structures of a program are maintained in memory associated with the general-

purpose core. Consequently, data must be copied to and from the special-purpose cores for every accelerated function. The time spent performing these extra memory copies may eliminate much of the performance gained by computing on a special-purpose core. This was the case in the version of Sweep3D that was written using the accelerator model [18].

In this paper, we have argued for a new approach to programming hybrid systems to overcome the shortcomings of the accelerator model. We call this approach the *reverse-acceleration model* because it represents the opposite view from the accelerator model of how one can program a hybrid system. While a general-purpose core drives the computation in the accelerator model, a special-purpose core drives the computation in the reverse-acceleration model. While a general-purpose core offloads compute-intensive functions in the accelerator model, a special-purpose core offloads control-, memory-, or I/O-intensive functions in the reverse-acceleration model. While general-purpose cores communicate with each other in the accelerator model, special-purpose cores communicate with each other in the reverse-acceleration model.

As a proof of the concept, we embodied the reverse-acceleration model in a messaging layer called the *Cell Messaging Layer* (CML), ported Sweep3D to it, and ran the code on Roadrunner, currently the world's largest hybrid supercomputer and fastest supercomputer of any type. Our results indicate that the reverse-acceleration model is a practical way to port non-hybrid codes to a hybrid system. The hybrid version of Sweep3D written to the reverse-acceleration model preserves the structure and much of the code from the non-hybrid version yet significantly outperforms the accelerator-model version of Sweep3D. We also demonstrated that the reverse-acceleration model has no inherent scalability limitations: Our version of Sweep3D performs well up to the 122,400 processor cores (general purpose plus special purpose) to which we had access.

Those seeking to port non-hybrid codes to hybrid systems should consider treating the system as a cluster of special-purpose processors with associated general-purpose processors instead of the other way around. The benefits of this reverse-acceleration model include ease of programming and, as we demonstrated for Sweep3D, superior performance—both important features as hybrid systems enter mainstream usage.

Acknowledgments

This work was funded in part by the Advanced Simulation and Computing program of the Department of Energy. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the U.S.

Department of Energy under contract DE-AC52-06NA25396.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of OpenMP Architecture Review Board Corporation, ClearSpeed Technology Corporation, Tiera Corporation, TOP500.org, Advanced Micro Devices, InfiniBand Trade Association, PCI-SIG Corporation, HyperTransport Technology Consortium, Broadcom Corporation, or Mellanox Technologies in the United States, other countries, or both.

***Cell Broadband Engine and Playstation are trademarks of Sony Computer Entertainment, Inc., in the United States, other countries, or both.

References

1. K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the Petaflop Era: The Architecture and Performance of Roadrunner," *Proceedings of the ACM/IEEE SC2008 Conference*, IEEE Press, Austin, TX, November 15–21, 2008; see <http://portal.acm.org/citation.cfm?id=1413372>.
2. J. Michalakes and M. Vachharajani, "GPU Acceleration of Numerical Weather Prediction," *Parallel Processing Lett.* **18**, No. 4, 531–548 (2008).
3. J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters," *Proceedings of the ACM/IEEE SC2008 Conference*, IEEE Press, Austin, TX, November 15–21, 2008; see <http://portal.acm.org/citation.cfm?id=1413379>.
4. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Mauerer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM J. Res. & Dev.* **49**, No. 4/5, 589–604 (2005).
5. K. J. Bowers, B. J. Albright, B. K. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 Pflop/s Trillion-Particle Particle-in-Cell Modeling of Laser Plasma Interactions on Roadrunner," *Proceedings of the ACM/IEEE SC2008 Conference*, IEEE Press, Austin, TX, November 15–21, 2008; see <http://portal.acm.org/citation.cfm?id=1413435>.
6. S. Swaminarayan, K. Kadav, T. C. Germann, and G. C. Fossom, "369 Tlop/s Molecular Dynamics Simulations on the Roadrunner General-Purpose Heterogeneous Supercomputer," *Proceedings of the ACM/IEEE SC2008 Conference*, IEEE Press, Austin, TX, November 15–21, 2008; see <http://portal.acm.org/citation.cfm?id=1413436>.
7. A. Kumar, G. Senthilkumar, M. Krishna, N. Jayam, P. K. Baruah, R. Sharma, A. Srinivasan, and S. Kapoor, "A Buffered-Mode MPI Implementation for the Cell BE Processor," Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds., *Proceedings of the 7th International Conference on Computational Science (ICCS 2007)*, Part I, Vol. 4487, *Lecture Notes in Computer Science*, Beijing, China, Springer, May 27–30, 2007, pp. 603–610.
8. M. Krishna, A. Kumar, N. Jayam, G. Senthilkumar, P. Baruah, R. Sharma, S. Kapoor, and A. Srinivasan, "A Synchronous Mode MPI Implementation on the Cell BE Architecture," I. Stojmenovic, R. K. Thulasiram, L. T. Yang, W. Jia, M. Guo, and R. Fernandes de Mello, Eds., *Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications (ISPA 2007)*, Vol. 4742, *Lecture Notes in Computer Science*, Niagara Falls, Canada, Springer, August 29–31, 2007, pp. 982–991.
9. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI Microtask for Programming the Cell Broadband Engine Processor," *IBM Syst. J.* **45**, No. 1, 85–102 (2006).
10. P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A Programming Model for the Cell BE Architecture," *Proceedings of the ACM/IEEE SC2006 Conference (SC'06)*, Tampa, FL, IEEE Press, November 11–17, 2006.
11. IBM Corporation, *Accelerated Library Framework Programmer's Guide and API Reference*, 2009. Publication number SC33-8333-03, product number 5724-S84, version 3, release 1.
12. Kevin O'Brien, Kathryn O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on Cell," *Intl. J. Parallel Programming* **36**, No. 3, 289–311 (2008).
13. L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Sci. Eng.* **5**, No. 1, 46–55 (1998).
14. J. Steed, W. Lundgren, and K. Barnes, "Gedae: A Tool for Implementing Software Radio on Heterogeneous Systems," *Proceedings of the 2004 Software Defined Radio Technical Conference and Product Exposition*, SDR Forum, Phoenix, AZ, November 15–18, 2004; see <http://www.gedae.com/documents/SDR%20-%20GEDAE.pdf>.
15. M. D. McCool, "Data-Parallel Programming on the Cell BE and the GPU Using the RapidMind Development Platform," *Proceedings of the GSPx Multicore Applications Conference*, Santa Clara, CA, October 31–November 2, 2006; see <http://www.rapidmind.net/pdfs/WPdpdm.pdf>.
16. K. R. Koch, R. S. Baker, and R. E. Alcouffe, "Solution of the First-Order Form of the 3-D Discrete Ordinates Equation on a Massively Parallel Processor," *Trans. Am. Nuclear Soc.* **65**, No. 108, 198–199 (1992).
17. D. J. Kerbyson and A. Hoisie, "A Performance Analysis of Two-Level Heterogeneous Systems on Wavefront Algorithms," E. John and J. Rubio, Eds., *Unique Chips and Systems*, Vol. 4, *Computer Engineering Series*, CRC Press, November 15, 2007, pp. 259–279.
18. F. Petrini, G. Fossom, J. Fernández, A. L. Varbanescu, M. Kistler, and M. Perrone, "Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine," *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, March 26–30, 2007.
19. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Trans. Graphics (TOG)* **27**, No. 3, 18:1–18:15 (2008).
20. S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, et al., "TILE64 Processor: A 64-core SoC with Mesh Interconnect," *2008 IEEE International Solid-State Circuits Conference (ISSCC), Digest of Technical Papers*, February 3–7, 2008, pp. 88–89, 598.
21. Top500 Organization, Top500 List, June and November 2008; see <http://www.top500.org/>.
22. C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron Processor for Multiprocessor Servers," *IEEE Micro* **23**, No. 2, 66–76 (2003).
23. P. Harvey, R. Mandrekar, Y. Zhou, J. Zheng, J. Maloney, S. Cain, K. Kawasaki, et al., "Packaging the Cell Broadband Engine Microprocessor for Supercomputer Applications," *Proceedings of the 58th Electronic Components and Technology Conference (ECTC)*, Lake Buena Vista, FL, May 27–30, 2008, pp. 1368–1371.
24. J. J. Dongarra, P. Luszczyk, and A. Petitet, "The LINPACK Benchmark: Past, Present and Future," *Concurrency and Computation: Practice and Experience*, **15**, No. 9, 803–820 (2003).
25. C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Trans. Computers* **C-34**, No. 10, 892–901 (1985).
26. H. Jin, T. Cortes, and R. Buyya, Eds., *An Introduction to the InfiniBand Architecture*, Chapter 42, G. F. Pfister, *High Performance Mass Storage and Parallel I/O: Technologies and*

- Applications*, Wiley Press and IEEE Press, November 26, 2001, pp. 617–632.
27. D. M. Pase and M. A. Eckl, “Performance of the AMD Opteron LS21 for IBM BladeCenter,” Technical Report, IBM Corporation, Research Triangle Park, North Carolina, August 21, 2006; see [ftp://ftp.software.ibm.com/eserver/benchmarks/wp_ls21_081506.pdf](http://ftp.software.ibm.com/eserver/benchmarks/wp_ls21_081506.pdf).
 28. T. Chen and D. A. Brokenshire, “BladeCenter QS: Maximizing Memory Performance,” Technical Report, IBM Corporation, July 1, 2008; see <http://www.ibm.com/developerworks/library/pa-qsmemperf/>.
 29. J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra, “The PlayStation 3 for High-Performance Scientific Computing,” *Computing Sci. Eng.* **10**, No. 3, 84–87 (2008).
 30. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, “The Potential of the Cell Processor for Scientific Computing,” *Proceedings of the 3rd Conference on Computing Frontiers*, Ischia, Italy, May 3–5, 2006, pp. 9–20.
 31. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, “Scientific Computing Kernels on the Cell Processor,” *Int. J. Parallel Programming* **35**, No. 3, 263–298 (2007).
 32. M. Kistler, M. Perrone, and F. Petrini, “Cell Multiprocessor Communication Network: Built for Speed,” *IEEE Micro* **26**, No. 3, 10–23 (2006).
 33. K. Chang, S. Pamarti, K. Kaviani, E. Alon, X. Shi, T. J. Chin, J. Shen, et al., “Clocking and Circuit Design for a Parallel I/O on a First-Generation CELL Processor,” *2005 IEEE International Solid-State Circuits Conference (ISSCC), Digest of Technical Papers*, San Francisco, CA, February 6–10, 2005, pp. 526–527, 615.
 34. D. Mayhew and V. Krishnan, “PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects,” *Proceedings of the 11th Symposium on High Performance Interconnects (HotI)*, Palo Alto, CA, August 20–22, 2003, pp. 21–29.
 35. S. Pakin, “Receiver-Initiated Message Passing over RDMA Networks,” *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, Miami, FL, April 14–18, 2008.
 36. A. K. Nanda, J. R. Moulie, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D’Amora, and S. Kesavarapu, “Cell/B.E. Blades: Building Blocks for Scalable, Real-Time, Interactive, and Digital Media Servers,” *IBM J. Res. & Dev.* **51**, No. 5, 573–582 (2007).
 37. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, Vol. 1, *The MPI Core*, 2nd edition, MIT Press, Cambridge, MA, September 1998.
 38. A. Nakano, R. K. Kalia, K. Nomura, A. Sharma, P. Vashishta, F. Shimojo, A. C. T. van Duin, et al., “De Novo Ultrascale Atomistic Simulations on High-End Parallel Supercomputers,” *Int. J. High Performance Computing Applic.* **22**, No. 1, 113–128 (2008).
 39. L. Peng, K. Nomura, T. Oyakawa, R. K. Kalia, A. Nakano, and P. Vashishta, “Parallel Lattice Boltzmann Flow Simulation on Emerging Multi-core Platforms,” *Proceedings of the 14th International Euro-Par Conference*, No. 5168, *Lecture Notes in Computer Science*, Las Palmas de Gran Canaria, Spain, Springer, August 26–29, 2008, pp. 763–777.
 40. H. Dursun, K. J. Barker, D. J. Kerbyson, and S. Pakin, “Application Profiling on Cell-Based Clusters,” *Workshop on Large-Scale Parallel Processing (LSPP), Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, May 29, 2009 (in press).
 41. A. Hoisie, O. Lubeck, and H. Wasserman, “Scalability Analysis of Multidimensional Wavefront Algorithms on Large-scale SMP Clusters,” *Proceedings of The 7th Symposium on the Frontiers of Massively Parallel Computation (Frontiers’99)*, Annapolis, MD, February 21–25, 1999, pp. 4–15.
 42. O. Lubeck, M. Lang, R. Srinivasan, and G. Johnson, “Implementation and Performance Modeling of Deterministic Particle Transport (Sweep3D) on the IBM Cell/B.E.,” *Scientific Programming* **17**, No. 2, 199–208 (2008).
 43. IBM Corporation, *C/C++ Language Extensions for Cell Broadband Engine Architecture*, February 27, 2008, Version 2.5; see <http://www.ibm.com/developerworks/power/cell/documents.html>.
 44. IBM Corporation, *Data Communication and Synchronization for Hybrid-x86 Programmer’s Guide and API Reference*, October 19, 2007, publication number SC33-8408-00, product number 5724-S84, version 3, release 0.
 45. R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, “Open MPI: A High-Performance, Heterogeneous MPI,” *Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar’06)*, Barcelona, Spain, September 25–28, 2006, pp. 1–9; see <http://www.open-mpi.org/papers/heteropar-2006/heteropar-2006-paper.pdf>.

Received November 7, 2008; accepted for publication May 21, 2009

Scott Pakin *Los Alamos National Laboratory, 30 Bikini Atoll Road, Los Alamos, New Mexico 87545 (pakin@lanl.gov)*. Since 2002, Dr. Pakin has worked as a Technical Staff Member in the Performance and Architecture Lab (PAL) at Los Alamos National Laboratory. His current research interests include analyzing and improving the performance of high-performance computing systems with particular emphasis on the communication subsystem. He has published papers on such topics as high-speed messaging layers, language design and implementation, job-scheduling algorithms, and resource-management systems. He received a B.S. degree in mathematics/computer science with Research Honors from Carnegie Mellon University in May 1992, an M.S. degree in computer science from the University of Illinois at Urbana–Champaign in January 1995, and a Ph.D. degree from the University of Illinois at Urbana–Champaign in October 2001.

Michael Lang *Los Alamos National Laboratory, 30 Bikini Atoll Road, Los Alamos, New Mexico 87545 (mlang@lanl.gov)*. Since 1999, Mr. Lang has been a Technical Staff Member at Los Alamos National Laboratory. For the last eight years, he has worked in the Performance and Architecture Lab (PAL). His current research interests include optimizing the performance of large-scale systems and investigating the performance of emerging high-performance computing architectures. His publications include system performance optimization and evaluation. He received his B.S. degree in computer engineering in 1988 and his M.S. degree in electrical engineering in 1993, both from the University of New Mexico.

Darren J. Kerbyson *Los Alamos National Laboratory, 30 Bikini Atoll Road, Los Alamos, New Mexico 87545 (djk@lanl.gov)*. Dr. Kerbyson is the leader of the Performance and Architecture Lab (PAL), a part of the Computer Sciences for HPC Systems group (CCS-1). He received his B.Sc. degree in computer systems engineering in 1988 and Ph.D. degree in computer science in 1993, both from the University of Warwick (U.K.). Before his appointment at the Los Alamos National Laboratory, he was a senior faculty member of computer science at the University of Warwick. His research interests include performance evaluation, performance modeling, and performance optimization of applications on high-performance systems as well as image analysis. He has published more than 100 papers in these areas over the last 15 years. He is a member of the IEEE Computer Society.