Transforming Scientific Codes to Execute Efficiently on the IBM Cell Processor as well as on Other Multicore Microprocessor CPUs

Paul R. Woodward, Jagan Jayaraj, and Pei-Hung Lin
Laboratory for Computational Science & Engineering
University of Minnesota
November 9, 2006

Introduction. The Cell processor, described on the IBM Web site a year ago in considerable detail, represents the first and the most extreme of a generation of multicore CPUs that is coming onto the market. For scientific codes that can rely mainly upon 32-bit arithmetical computations and that can be formulated in terms of vector computing concepts, Cell is the most rewarding of this new generation of CPUs, so far as we are presently aware. Since we are not hardware design experts, nor are we compiler experts, it is not completely clear to us what specific features of Cell are responsible for its very high performance. By comparing Cell SPU (synergisitic processing unit) performance with that of other microprocessors, it seems that the most important performance features of Cell are the large number of SPU vector engines on each CPU chip and the ability of the SPU to perform 4 multiply-adds per clock cycle, as compared to just 2 for the Pentium-4 vector hardware, for example. The Cell SPU's large number of quadword registers and its asynchronous DMAs should also be playing important roles in its delivered performance on our application, but we do not have clear evidence of this at this writing.

Running our entire PPM gas dynamics code on two Cell processors sharing a common memory on a single blade system, we find that 1040 time step updates of a single grid brick of 128^3 cells are performed in just 63 seconds. Each such time step update requires 5.07 Gflops, and therefore the aggregate performance of these 2 Cell processors is 83.7 Gflop/s. These Cell processors, made available to us through the assistance ofStephen Hodson, Robert Lowrie, and Ben Bergen at Los Alamos, were running at 3.2 GHz. The performance of each individual SPU on this code is thus 5.23 Gflop/s and the aggregate performance of the 8 SPUs on a single Cell CPU to 41.8 Gflop/s, or 21% of the peak performance of the 8 SPUs.

To put the above performance measurements into perspective, we have measured the performance of the identical PPM gas dynamics code implementation on standard Intel CPUs. To run on these CPUs we have used an expression in standard Fortran 77 with vectorization directives for the Intel Fortran compiler. The Intel Xeon and Pentium-M serve to illustrate that the code transformations for Cell that will be discussed below produce performance benefits on all microprocessors. Both of these micro-processors, which are not of Intel's latest, Core-Duo design, are able to perform only 2 operations per clock tick in 32-bit SIMD mode. For the 3.6 GHz Xeon, the 2 operations can be multiply-adds, so that its peak performance rating in 32-bit mode is 4 times its clock frequency, or 14.4 Gflop/s. For the 2.16 GHz Pentium-M, multiply-add instructions are not implemented, so that the peak performance is only double the clock frequency, or 4.32 Gflop/s. The performance of the Fortran PPM code, as transformed for Cell, on the Pentium-M is 2.15 Gflop/s, or 50% of peak. For the Xeon CPU, the performance is 3.3 Gflop/s, or 23% of peak. These performance figures indicate that the multiply-add instructions offered by the Xeon and the Cell SPU are of essentially no use to us. This fact was already clear to us from the manual implementation in C plus intrinsics for Cell, where we were able to find essentially no opportunities to use these instructions. The performance figures also show that we see the same percentage of peak performance on the Xeon and the Cell SPU. We may therefore tentitavely conclude that the DMA capability and the large number of registers on the SPU are of no particular assistance in this gas dynamics application. Testing this code on the newer Intel 5100 series ("Woodcrest") Xeons, which run at 3.0 GHz and which, like the Cell SPU, can perform 4 multiply-adds per clock tick in 32-bit mode gives the following results. When all the data updated in the fluid flow problem fits into the Xeon CPU's 2 MB cache, delivered performance is 6.01 Gflop/s/core. When this data does not fit into the cache, performance drops to 5.48 Gflop/s/core. When 4 of these Xeon cores run this code simultaneously from the same PC memory, performance per core drops 5% further to 5.26 Gflop/s/core. This matches the 5.23 Gflop/s/core performance of the 3.2 GHz Cell processor very closely, but of course the Cell processor has 4 times as many such cores as the Intel Woodcrest CPU.

At the moment, it is clear from the above performance measurements that the Cell processor, with its 8 vector engines, holds the present performance record on the PPM gas dynamics application, and by a wide margin. It is therefore worth considerable effort to implement a code on this new CPU. The most fundamental challenge in running a code on the Cell processor is to squeeze the application into the 256 KB local store of the SPU. This requires rather extreme code transformations. These are not required on other CPUs, which have 8 times the local on-chip memory, namely 2 MB/core. However, we will see that these transformations nevertheless increase the code performance and are therefore a good idea regardless of the amount of on-chip memory available. They also tend to deliver very high performance on very small chunks of the simulation domain, which makes highly efficient implementations on extremely large parallel systems practical without unduly increasing the grid size. In the following sections these code transformations are discussed in detail.

Transformation of the PPM gas dynamics algorithm's implementation for Cell. The diagrams at the top of the next page illustrate the fashion in which the PPM gas dynamics computation [1-8] is reorganized for efficient execution on the Cell processor SPU. First, the principal physical state variables – the density, pressure, 3 components of velocity, and a passively advected ink or dye that is used in many PPM simulations – are organized in the main memory in units of tiny grid chunks of 4 grid cells on a side, as shown. A single, multidimensional Fortran array, DD, is dimensioned as follows:

```
dimension   DD (1:4, 1:4, 1:4, 1:6, 0:nbx+1, 0:nby+1, 0:nbz+1)
```

For the x-pass of the algorithm, the first 3 of these array dimensions are assigned to y, z, and x, respectively, so that the 2 fast-running indices go over 16 independent grid strips in the direction of this x-pass. The fourth index is over the 6 physical state variables, and the last three indices go over the x, y, and z, coordinates of the grid chunk within a larger grid brick, including one ghost chunk at each end of every strip of chunks. With 32-bit arithmetic, which is all the PPM gas dynamics algorithm has ever required, each chunk, with all 6 of its physical variables, amounts to just 1.5 KB of storage. This chunk record of 384 words (1.5 KB) becomes the quantum of main memory storage that is dealt with by the PPM implementation. Therefore, from the point of view of many sections of the code, we need not consider the substructure of this record, and we may, in Fortran, equivalence DD to DDb, dimensioned as

```
dimension   DDb (1:384, 0:nbx+1, 0:nby+1, 0:nbz+1)
```

The 16 SPUs of the 2 Cell processors sharing a common memory on an IBM dual-Cell blade system can cooperatively update a grid brick of 128^3 cells, so that we have $nbx = nby = nbz = 32$. Each SPU will update 64 strips of 32 chunks in each 1-D pass of the PPM algorithm. The work of each SPU and all the data it operates on is completely independent during such a 1-D pass. If we consider the program that will reside in the SPU's local store throughout this process, it will contain a loop over its 64 strips of chunks and inside that a loop over the chunks of each strip. It is the body of this inner loop, which is built to operate upon a single grid chunk, that concerns us, since its structure is quite unusual. One iteration of this inner loop is outlined below.

1. When we execute this inner loop body for the first time for a grid strip, we first fetch the zeroth (ghost) chunk record, and then we immediately set going an asynchronous DMA to prefetch the first chunk record. This will keep data streaming into the SPU local store, so that data will always be there when we need it. At the end of this loop, if we have produced a new, updated chunk record, we will set going an asynchronous DMA to write that back to main memory just before we return to do the next loop iteration that processes the next chunk in our strip.

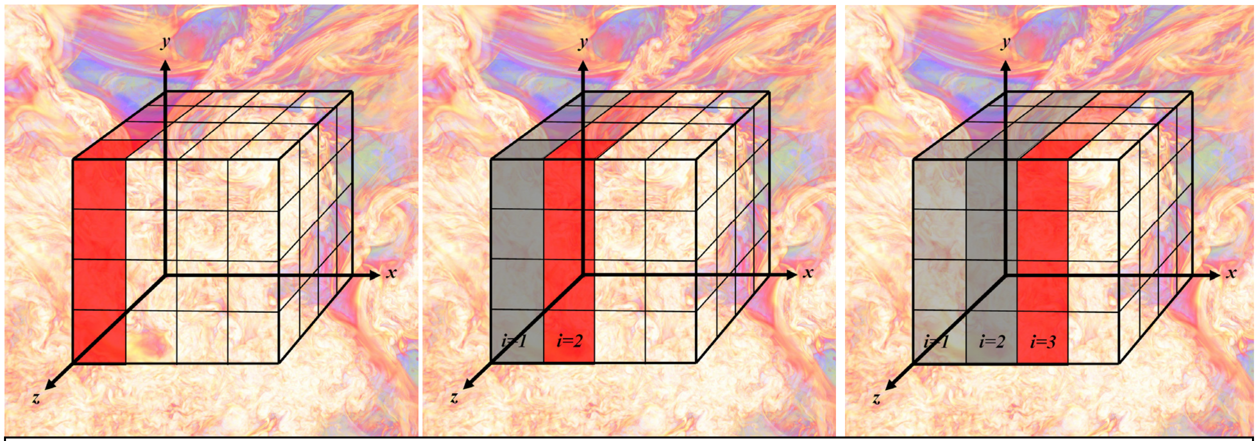


Figure 1. Illustration of the data structure, vectorization strategy, and loop fusion technique used for PPM.

2. Now we enter a loop over the 4 planes of cells in our grid chunk. One such plane is highlighted in red in the diagram at the left in the figure. Each column of cells in this grid plane will correspond to a quadword data structure, an intrinsic data type, for any variable we are working with. Four of these quadwords will constitute the information about this variable for any particular grid plane. Grid planes of values will be the fundamental working units of our fully vectorized computations. Each operation we perform will consist of 4 independent quadword operations. This can be expressed in the extended C language for the SPU, and it can also be expressed as a vectorizable loop in Fortran over the 16 fast-running index elements of every data object we will reference (with the exception of literals, constants, and vector temporaries, which the compiler will expand for us).

3. Working data structures: The index of our loop over grid planes in our chunk is i, as shown in the figure. Each of our variables will have a different range of i values for which we must remember it. This range will be denoted by a second, slow-running Fortran array index. The particular value of this index for each iteration of our loop over i will be given by an integer variable that will act like a pointer. This will allow us to reference the plane of values for the present i iteration as well as those for previous iterations without having to copy data unnecessarily. For the fundamental physical state variables, we will need to remember 5 planes of values. Some variables will require that we retain only 4 planes, others 3, etc. Holding only the values that we require complicates the variable indexing, but it saves a great deal of space in our SPU's local store. Even on a cache-based CPU, where we might have as much as 2 MB of space for our data, using this indexing scheme is preferred, since it will minimize any pointless writing back of this working data to main memory.

4. In the first inner-inner loop, which ranges over the 16 elements for a 4×4 plane of cells, we do all the work that the algorithm requires and that is possible with the cell-based data that we have read in from our chunk record. For PPM, this includes evaluating cell-averaged estimates of sound speeds, characteristic speeds, the Courant number, and the like. This portion of the computation is indicated by the left-most diagram in the figure, where only the first plane of cells in our chunk is highlighted. At the outset of this inner-inner loop, we copy a single plane of data values out of the chunk record for each of the fundamental physical state variables contained in the record. These will have been stored in precisely the format we need for this 1-D pass, so that this unpacking of the data record is extremely fast.

5. If this is the first iteration of our loop and this is the first chunk in our grid strip, we must now jump to the end of our loop over grid planes, since we do not yet have sufficient data generated to proceed to the second inner-inner loop.

6. If not, we have remembered the results from the computation in the first inner-inner loop on our previous iteration, and we therefore have values generated for our present plane of grid cells and for the plane to the left of it, as shown in the center diagram of the figure. In this case, we now execute a second loop over the 16 elements in a plane of cell-based values and we compute a whole variety of quantities that refer to the interfaces between the cells of our present grid plane and their neighbors on the left. For the PPM algorithm, these quantities are mostly needed in the interpolation process to determine parabolas to represent Riemann invariants inside grid cells.
7. If this is the second iteration of our loop for the first chunk in our grid strip, then we must jump to the end of our loop over grid planes, because to proceed to the next inner-inner loop we require quantities evaluated for both left- and right-hand interfaces of grid cells.
8. If not, we proceed to the next inner-inner loop. In this loop we will generally evaluate cell-based quantities using both left- and right-hand interface values for these cells. This plane of cells must therefore be one plane further to the left than the one pointed to by our loop index i.
9. We proceed in this fashion, performing one inner-inner loop after another, alternating between the computation of cell-based and interface-based quantities. For the simplified PPM algorithm (it is simplified, but it is not sPPM) that we have implemented for Cell, there are 9 such inner-inner loops, each over 16 elements and fully vectorizable on the SPU or on any other microprocessor manufactured today. Because we have 9 of these loops, we must process 2 entire grid chunks before we can generate any updated values of the fundamental physical variables at all. When we process the third or greater chunk in our strip, we get one plane of new physical variable values on each loop iteration. These we pack into a new chunk data record that we hold in our SPU's local store. We pack this data so that it will be formatted as required for the next 1-D pass. For 4 out of the 6 1-D passes in a symmetrized pair of time step updates, we will need to transpose this data as we put it into the new chunk record. This is of course no problem, since we are working inside our local store where there are no difficulties associated with cache lines. Once our new chunk record is complete, at the end of the processing for our present chunk, we set off the asynchronous DMA that will write it back to main memory.

The performance of this transformed code on the Cell processor blade has been given earlier. We may understand this performance from the following considerations. First, we can see that the bandwidth between the Cell processor's 8 SPUs and their shared main memory is sufficient to support all 8 of them running our PPM code at 5.23 Gflop/s @ 3.2 GHz simultaneously. This provides an aggregate performance for the 2 Cell processors on a single blade of 83.7 Gflop/s. To process each SPU's 64 strips of 32 real and 2 ghost grid chunks requires each SPU to perform 758 flops per real (not ghost) grid cell. The cost of all ghost cell computations is included in this figure of 758 flops/cell. Thus to perform a single 1-D pass, each SPU does 99.3 Mflops, and at 5.23 Gflop/s this takes 18.9 msec. During this time interval of 18.9 msec we must read into each Cell processor 512×34 grid chunk records of 1.5 KB each, and we must write out 512×32 such records. Therefore we will need to simultaneously stream in data at 1.35 GB/sec and stream out data at 1.27 GB/sec. Since 25.6 GB/sec of bandwidth is available in each direction for each Cell processor, this is no problem, and hence introduces essentially no additional cost. The memory system on the Dell dual dual-core 3.0 GHz Intel Woodcrest Xeon machine has about half this memory bandwidth and half the number of processor cores, and the performance of each core on this code drops from an in-cache value of 6.01 Gflop/s/core to 5.26 Gflop/s/core. The benefits of this code transformation are therefore quite general across commodity CPUs manufactured today.

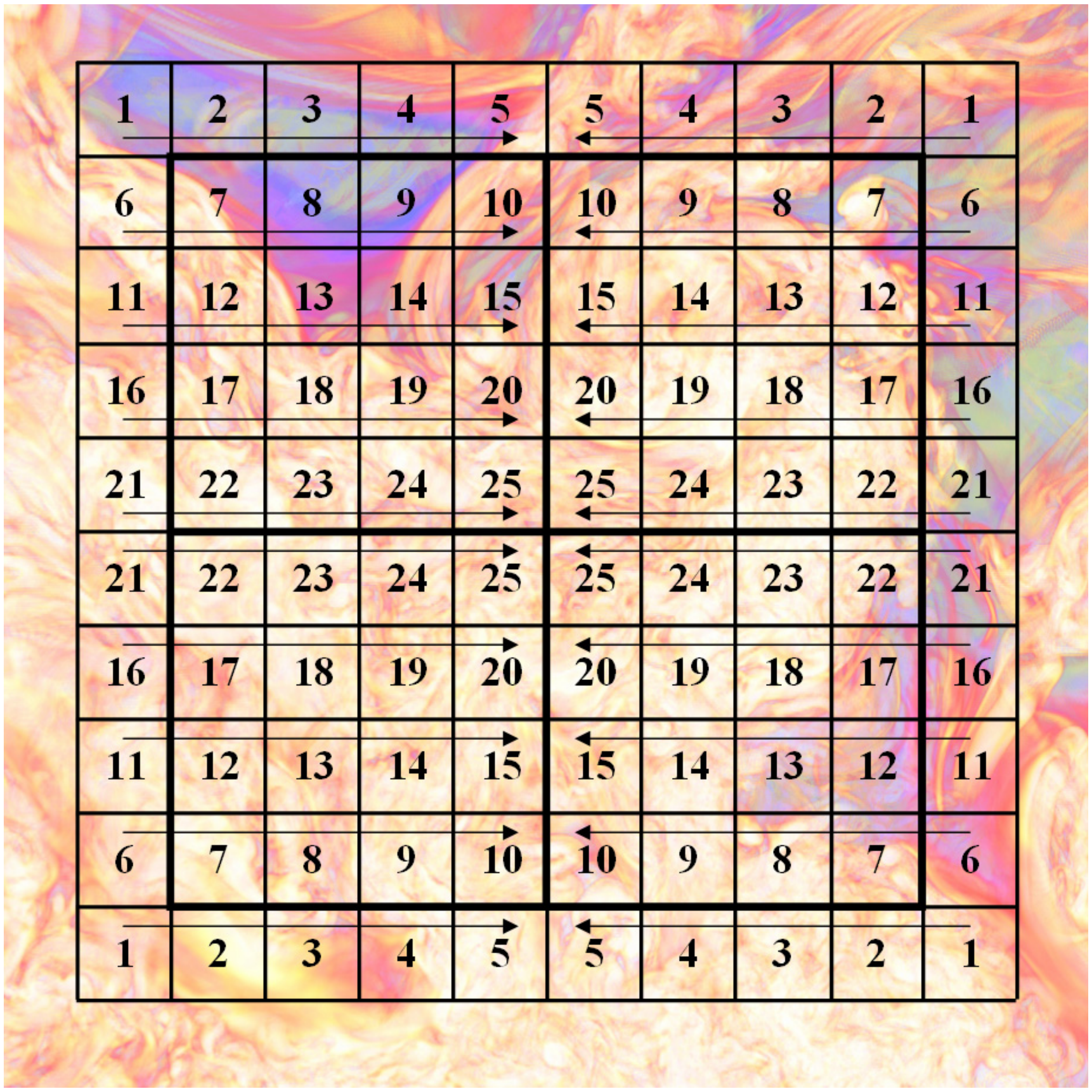
The PPM implementation that we tested on the dual Cell blade at Los Alamos was written to update a grid brick of 128^3 cells in a manner that is appropriate if this grid brick is only one of a large number of such bricks in a single problem. We begin each time step update with an augmented brick consisting of 32^3 real chunks plus a layer one chunk (a ghost chunk) thick all around. In the first 1-D pass, we update 34^2 strips of 32 real chunks each. Thus 4 of the 16 SPUs are updating their 73rd strips of chunks while the remaining 12 SPUs are idle, waiting for them to finish up this 1-D pass. In the second 1-D pass, we update 32×34 strips of 32 real chunks, and in the final 1-D pass, we update 32^2 such strips. Thus in both these last 2 passes, we have perfect load balancing. For the entire time step, the number of chunk strip updates is 73+68+64 = 205, with all but one of these involving all 16 SPUs. The cost of the 12 idle SPUs in this single chunk strip update is therefore less than ½%. With a 3.2 GHz clock, each such time step will take (205/64)×18.9 msec = 0.0607 sec. In a parallel implementation in which each dual Cell blade owns, for example, 4 grid bricks, each taking up only 57.6 MB of its 1 GB local memory, the blade would need to fetch (34^3–32^3) chunks and send this same number over the network in the time it requires to update a single grid brick for one time step. This requires streaming 9.57 MB in and 9.57 MB out of the blade over the network every 0.0607 sec. Therefore, this requires a network bandwidth of just 315 MB/sec, hardly a demanding requirement on an Infiniband network. If we place only a single grid brick on each blade, we can transmit two chunks of data upon each update for a 1-D pass of a strip of 32 chunks. This would generate 32 sends of 1.5 KB chunk records every 18.5 msec, and of course the same number of such messages would be arriving in the same time interval. The bandwidth requirement for the network interconnect is ever so slightly lower in this case, but the latency requirement is more difficult to meet. Nevertheless, a cluster of dual Cell blades could work at the same very high performance level on a 4 times smaller problem. If we reduce the size of the single grid brick to just 64^3 grid cells, making the problem domain 8 times smaller, we require double the network bandwidth and a two times smaller latency to compute at the same level of performance. However, this might still be achievable on an Infiniband network, even at single data rate (SDR). In this case, each time step would take only 7.26 msec. A grid of 512^3 cells would require only 512 dual-Cell blades, and the 10,000 time steps that such a run might require would take only a minute and 13 seconds. This run would almost be over before we could look away, take a sip of coffee, and look back at our graphic display! But of course no proud owner of such a system would ever allow us to do such a trivial problem on it.

The code transformation procedure outlined above is quite unusual, but it is built from concepts that have been known for quite some time. The idea of doing all computations in the numerical algorithm at once in order to save memory utilization and boost performance has been incorporated into all PPM variants since 1980, when the Cray-1's high performance on relatively short vectors and its use of vector registers made this approach possible. However, all PPM codes to date have used the 1-D pass update of a single grid strip as the computational unit into which all the code's physics was combined. The fusion of all this computation for a single chunk of such a grid strip is relatively new. The construction of the loop over grid planes that is outlined above is a derivative of the process of aggressive loop fusion that our team developed nearly a decade ago for the sPPM benchmark code for the use of vendors whose CPUs performed better at scalar arithmetic than at vector arithmetic [6]. The old MIPS and PA RISC CPUs fell into this category, exhibiting speed-ups of 58% and 100% from this code transformation, respectively, when the resulting single scalar loop was unrolled once. All other CPUs of that era delivered very near to top performance on such a transformed code. Here we have adapted this technique to the Cell processor's SPU by adding at each step a vectorizable loop over 16 independent grid strips. This makes the memory footprint 16 times larger, but the footprint is still small (namely 25 KB) and the computational performance is very, very much greater on today's CPUs. Over a few months, we have experimented with a fairly large number of possible implementations, running them not only on the SPU emulator, but also on standard Intel CPUs and on teams of processors over Infiniband networks and on such AMD CPU teams within a Cray XT3 machine. Our experience shows that the above code transformation is optimal, at least for our codes, and it is optimal on all platforms we have yet tried.

We note that, supported by NSF's PACI program through NCSA, we used a variant of this Fortran code transformation a few years ago to accelerate the performance of Bob Wilhelmson's NCOMMAS meteorology code, achieving a factor of 5 speed-up of the complex advection scheme in the code in taking it over to the early Intel Itanium CPUs. The compiler team at Rice was able to automate this code transformation, but a true precompiler for it was never built. The principle benefit of this code restructuring is to keep the local memory footprint of an algorithm small and thus to allow reuse of data that is in a cache or local memory. For Cell's SPUs, this feature is critical, since the local store is much smaller than the cache memories we have become accustomed to over the last decade of microprocessor designs. Above, we have used this transformation to implement a 1-D pass of a very complicated numerical algorithm. The translated code is 80 KB, when its inner-inner loops are unrolled 4 times, and its working data context is 25 KB, so that both fit easily together in the SPU's 256 KB local store. For algorithms that are explicitly multidimensional, like the NCOMMAS advection scheme, the transformation may result in a significantly larger local memory footprint. Nevertheless, such algorithms can be successfully restructured for Cell. As an example, we describe the implementation of a 3-D advection scheme, PPB (piecewise-parabolic Boltzmann scheme), that was introduced in 1986 [4] based upon earlier work of van Leer [9] and which has been streamlined and enhanced for highly efficient parallel computation [10]. Like the advection scheme used in Wilhelmson's code, this is built out of 1-D passes, but there are three times fewer of them, the scheme is far more accurate, and the scheme runs at a much larger time step (any Courant number below unity). This PPB method has been used for several years in our PPM codes to track the fractional volumes of multiple distinct fluids, and it is implemented in the new PPM module released last summer with the Los Alamos XRAGE code.

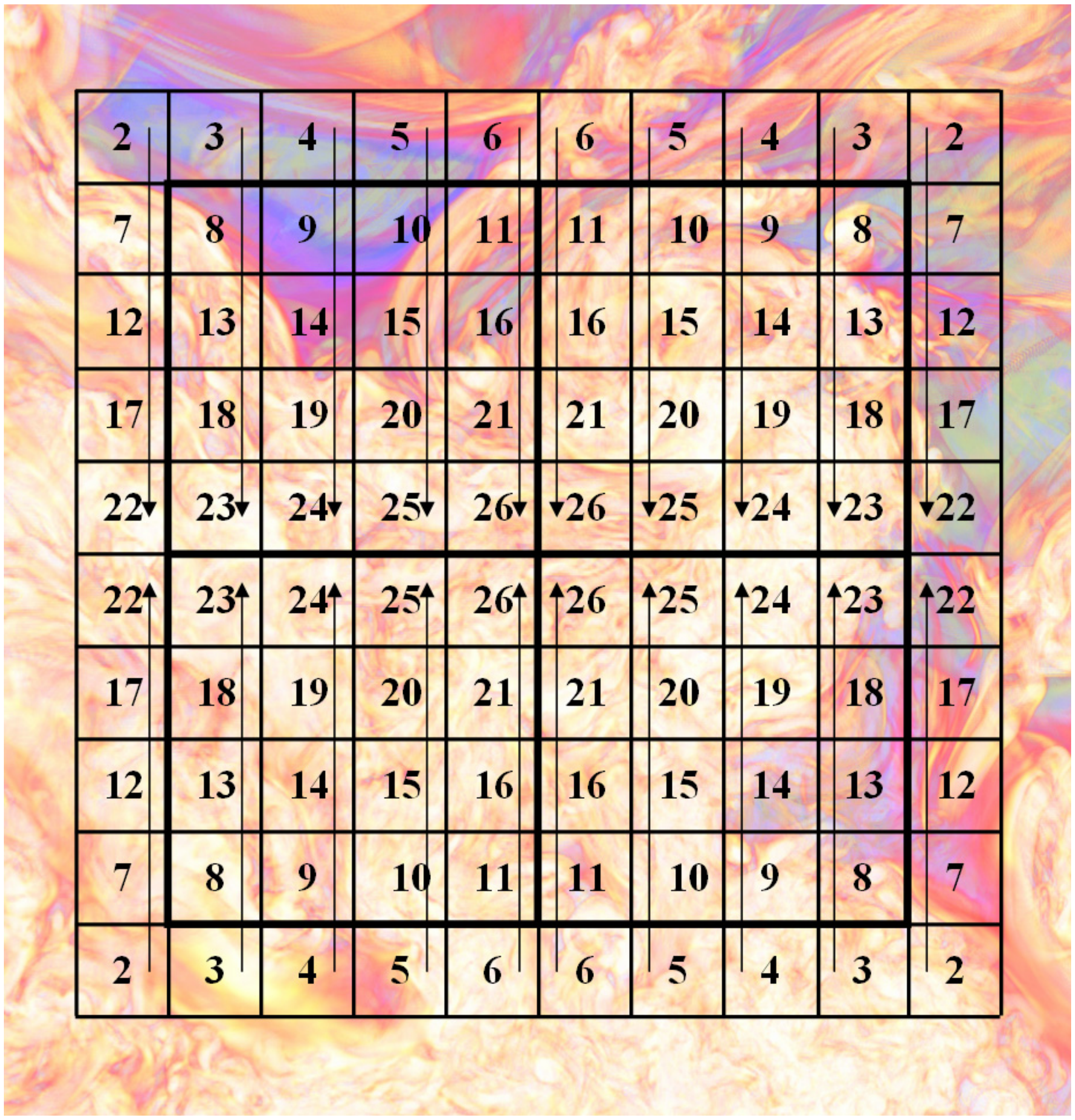
Transformation of the PPB advection algorithm's implementation for Cell. The implementation of a 1-D pass of the PPB algorithm for Cell is quite similar to that described above for PPM. The principal difference, however, comes in considerations of memory bandwidth. A 1-D pass for PPB performs just 281 flops per grid cell update, with an overhead for ghost cell processing that is 266 flops per grid strip. This means that in updating a strip of 32 cells, 289 flops/cell rather than 281 must be performed. This overhead is relatively low, compared to the corresponding overhead for PPM, because the difference stencil for a PPB 1-D pass consists of the cell to be updated plus just 2 cells on either side. To get efficient vector performance, we implement this algorithm to operate on grid chunks of 4^3 cells, just like PPM. Because of the smaller difference stencil, we could, in principle, perform 2 1-D pass updates in immediate succession on each chunk while it is in the cache memory, but this would lead to inaccuracy when the advection velocity has significant spatial variation. It would cause the gas to be advected in each direction for up to 2 cell widths before being advected back to its proper path in the other 1-D passes. If the velocity field is uniform, this leads to no additional error, but if it is nonuniform (It "always" is), this would lead to advection in slightly wrong directions. The advantage of the double 1-D pass is that it would improve the computational intensity of this algorithm. A single 1-D pass requires us to read in from main memory 13 (32-bit) words of data and to write back 13 for each cell that we update. We then perform only 10.8 flops/word, which is about 1/6 of the computational intensity of the PPM algorithm. To keep this algorithm going at its top speed would therefore require about 6 times the main memory bandwidth needed to feed PPM. PPB has been transformed for Cell in Fortran, but because the Cell processor does not yet understand Fortran, this code has so far been tried only on Intel CPUs. Our experience with PPM indicates that Cell SPU performance is likely to be similar, although for this algorithm Cell's DMAs might have a considerable positive performance impact.

Implementing the PPB 1-D pass in Fortran for Cell, in the manner described for PPM above, and compiling it with the Intel 9.1 Fortran compiler and running it on a Dell workstation with two dual-core 3.0 GHz Woodcrest Xeon CPUs sharing a 533 MHz DDR memory of 16 GB, we obtain the following performance for updating in succession, chunk by chunk, bundles of grid strips of $4{\times}4{\times}N$ grid cells so that we ultimately update grid bricks of N^3 cells. Running this code on just a single such core for $N = 8$, 16, 32, 64, and 128, the delivered performance is 4.44, 4.67, 4.43, 3.17, and 3.01 Gflop/s respectively. We see that the performance drops down when the entire grid brick of N^3 cells no longer fits into the cache memory of 2 MB. For the 3.8 GHz Intel Pentium-4 CPU, results of this same test are 2.16, 2.30, 2.03, 2.02, and 2.00 Gflop/s, so that we see that the memory can keep up with the needs of this slower processor core. Modifying this code to do the dual 1-D pass update discussed above, and thus doubling the algorithm's computational intensity, give performance measurements on the Pentium-4 of 2.16, 2.32, 2.21, 2.23, and 2.18 Gflop/s, so that we clearly see that the memory bandwidth is not limiting the performance here. However, running this dual 1-D pass update code on the 3.0 GHz Woodcrest CPU



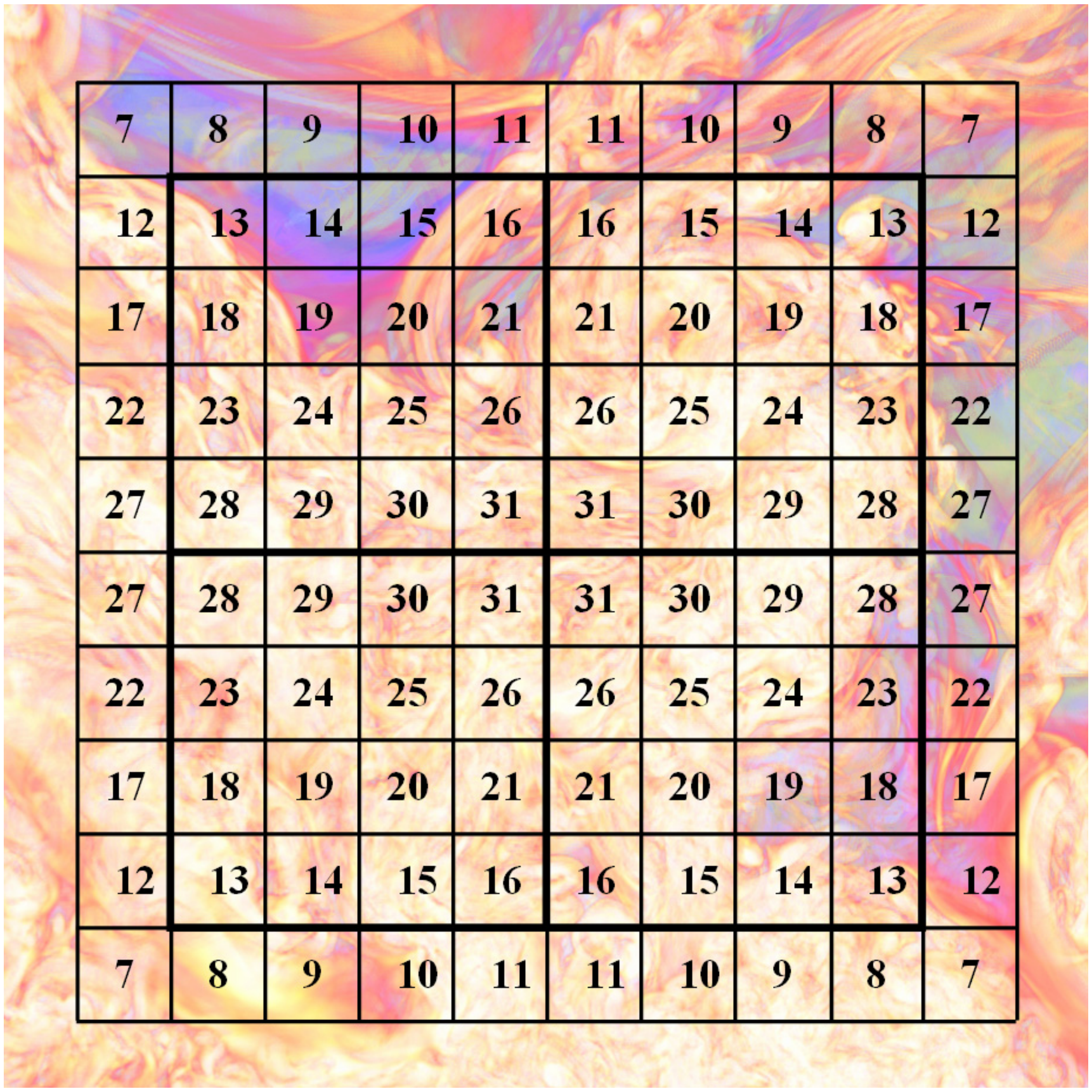
core gives 4.48, 4.64, 4.62, 3.78, and 3.63 Gflop/s. Performance on the last, thoroughly out-of-cache problem updating the 128^3 grid brick has increased by 20% at this computational intensity level of 21.6 flops/word. Modifying the code again to do 3 1-D pass updates brings this performance up still another 10% to: 4.44, 4.60, 4.67, 4.06, and 4.02 Gflop/s. We have said that this approach at increasing the computational intensity of the PPB algorithm is not practically useful, but it is instructive. It tells us that if we can perform 3 1-D pass updates on a grid chunk of data for this algorithm while that data resides in cache, we will obtain roughly 86% of the performance that the vector engine can deliver to a memory of infinite bandwidth. But to do this, we must perform the 3 1-D passes in the 3 different dimensions, x, y, and z. This is a challenge, especially with the very small local store of the Cell SPU. The procedure for meeting this challenge, described below, is completely general, and it will work for any truly multi-dimensional numerical algorithm. It will, of course, also work on any CPU manufactured today, but Cell will, at least today, deliver the greatest benefit for this code restructuring.

The implementation of the 3-D PPB advection algorithm for Cell is illustrated by the diagrams on this and succeeding pages. Each square in the diagram represents a single chunk of 4^3 grid cells, corresponding to a data record of 3.25 KB. The bold lines demarcate the regions that are updated by each of 4 SPUs in the Cell processor, and ghost cell grid chunks surround the region to be updated. In the



diagram, we show only a single plane of a grid brick consisting of 8^3 grid chunks, or of 32^3 grid cells, with the ghost cell chunks all around. The four Cell SPUs update planes of such grid chunks in succession, beginning either at the far or the near face of the grid brick in the z-dimension. Two groups of 4 SPUs each perform such updates, workimg from the outer faces of the grid brick inward, one plane of chunks at a time, until they meet in the center. Thus the diagrams actually represent the action of all 8 SPUs, by symmetry.

The arrows in the diagrams represent the direction of the 1-D sweeps of the algorithm – x for the first diagram, y for the second, and z, with no arrows (they would point either out of or into the plane of the diagram), for the last. The numbers in the squares in these diagrams give the "episode" number. Each numbered episode of computation involves the processing of one grid chunk in, potentially, each of the 3 dimensional sweeps. Epsiode 1 involves only a single x-pass chunk update for each of the 8 SPUs, since we cannot proceed to do any y- or z-pass updates without first obtaining fully updated information from the x-pass. This first x-pass chunk update is part of the "priming of the pump" in this pipelined operation. Note that 4 of the 8 SPUs sweep in the positive and 4 in the negative x-direction. This means that we need to program two versions of our x-pass code, one for sweeping in each of these two opposite



directions. This of course is easily done. Just as with the PPM algorithm, the processing of each grid chunk for a 1-D pass results in an active data context of arrays in the SPU local store that are utilized in the next chunk update. For PPB, this active data context consists of 103 planes of data, or 6.44 KB. When the pairs of SPUs that are jointly updating each row of chunks in the diagram meet in the center of the row, they must exchange this active data context over the Cell processor's 200 GB/sec element interconnect bus (EIB) in order to complete the update for the last grid chunk right next to the center without further, redundant computation. This data exchange happens, for the x-pass updates, in episodes 5, 10, 15, etc.

The second diagram illustrates the y-pass of the PPB algorithm, which of course runs the identical program as the x-pass but operates upon data that has been transposed in the local store at the end of the x-pass update. The episode numbers given refer to the ends of the x-pass portions of the indicated episodes. Thus, after the grid chunk updates for the x-pass in episode 3 are completed, we have new grid chunk data records (of 3.25 KB) ready to be processed for the y-pass in the grid chunks labelled by episode number 3 in the y-pass diagram. (If we wanted to run this algorithm for a complete pair of 3-D time step updates, that is, for 6 1-D passes, we have enough ghost cell data available, but we would need to perform the y-pass chunk updates designated for ghost cell chunks in the second diagram.) In the third diagram, showing the z-pass operations performed in each episode, we see that no z-pass processing can begin until the end of the y-pass updates in episode 7. If we are doing only a single time step update of 3 1-D passes, we would not need to update this ghost cell grid chunk, and then we would do no z-pass processing until the end of the y-pass updates in episode 13. It therefore clearly takes us a long time to get this pipelineed process fully underway. All the time that we are filling the pipeline, we need to be saving in the SPU local stores active memory contexts and/or grid chunk data records, which can add up to a substantial data volume. However, by episode 13, our pipeline if full, and the processing is tremendously efficient. Because we do not process chunks in the z-pass in z-direction order, we must save an entire plane of active data contexts for the z-pass processing of 6.44 KB each. The y-pass processing also is not in the preferred order, and this requires that we save in the SPU local store an entire row of y-pass active data contexts, again of 6.44 KB each. For the update of the 32^3 cell grid brick that is illustrated in these diagrams, each of the 8 SPUs must save in its local store 16+5+1 = 22 active data contexts, which amounts to 142 KB of data. Happily, this fits easily into the 256 KB local store beside the program itself, which is very small indeed, since it does so little work in each chunk update. Once the pipeline is filled, we read in one grid chunk record of 3.25 KB at the outset of every episode (we prefetch one, that is, that we will not need until the next episode), and we write back one completely updated chunk record of 3.25 KB. For the example in the diagrams, and for an update of only one 3-D time step, we read in 1000 chunk records and write back 512. This amounts to $3 \times 10.8 \times 512/1000 = 16.6$ flops/word, which is aonly a 60% increase in computational intensity. If the SPU local store were 4 times larger, we could update a brick of 64^3 cells, with a computational intensity of $3 \times 10.8 \times 4096/5832 = 22.8$ flops/word. The painfully slow increase in the computational intensity for this algorithm is just life. Not every algorithm has as much work to perform on its data as PPM. However, if we can combine the relatively low intensity work of PPB with the high intensity work of PPM in a single multifluid gas dynamics operator (and of course this is exactly what we do actually need to do in a multifluid code), then the overall computational intensity can still be high. We will have larger active data contexts in the SPU local stores and a larger code size there also, but both will still fit easily. Such an implementation has been undertaken but is not complete at this writing, since the process of translation from Fortran to C + intrinsics for the SPU is so arduous at this time.

Numerical algorithms that are still much less computationally intensive than even PPB advection abound in the computational science community. As an example, we outline how we would implement the popular red-black relaxation scheme for the Dirichlet or Poisson problem. This implementation could then be used as an element in a full multigrid relaxation computation for the Poisson problem. Our technique will be similar. We will bring into our local memory as small a chunk of new data as permits efficient vector computation, and then we will do every useful thing we can think of with this data before writing results back to main memory.

A red-black relaxation scheme for a multigrid Poisson solver. In the diagram at the left on the next page, the 32 new black square values are shown that are read into local memory in the first step of the 2-D red-black relaxation algorithm. These new black-cell values at iteration 0 are then used to compute 32 new red-cell values at iteration 0 as shown. Each new red-cell value is of course the average of its nearest black-cell neighbor values, plus the red-cell value of the source term. The 17 white cells designate black-cell values at iteration 0 that must be remembered from the previous sequence of steps in order to perform this computation. In the diagram at the right, the 32 new red-cell values at iteration 0 that are produced in step 1 are used to compute the 32 new black-cell values at iteration 1 that are shown. The 16 white cells designate the red-cell values at iteration 0 that we must remember from the previous sequence of steps in order to perform this second step of the computation. Each time this set of 2 steps is repeated, we end up with 32 new black-cell values at one greater iteration level and in the pattern shown, but this pattern is translated downward two rows and to the left two columns of grid cells. If we do these two steps 4 times in succession, we end up with a block of 32 new black-cell values at iteration level 4. These new black-cell values are located 8 rows down and 8 columns to the left of the block of 32 old black-cell values that we read in from memory before executing the first step of this 8-step sequence.

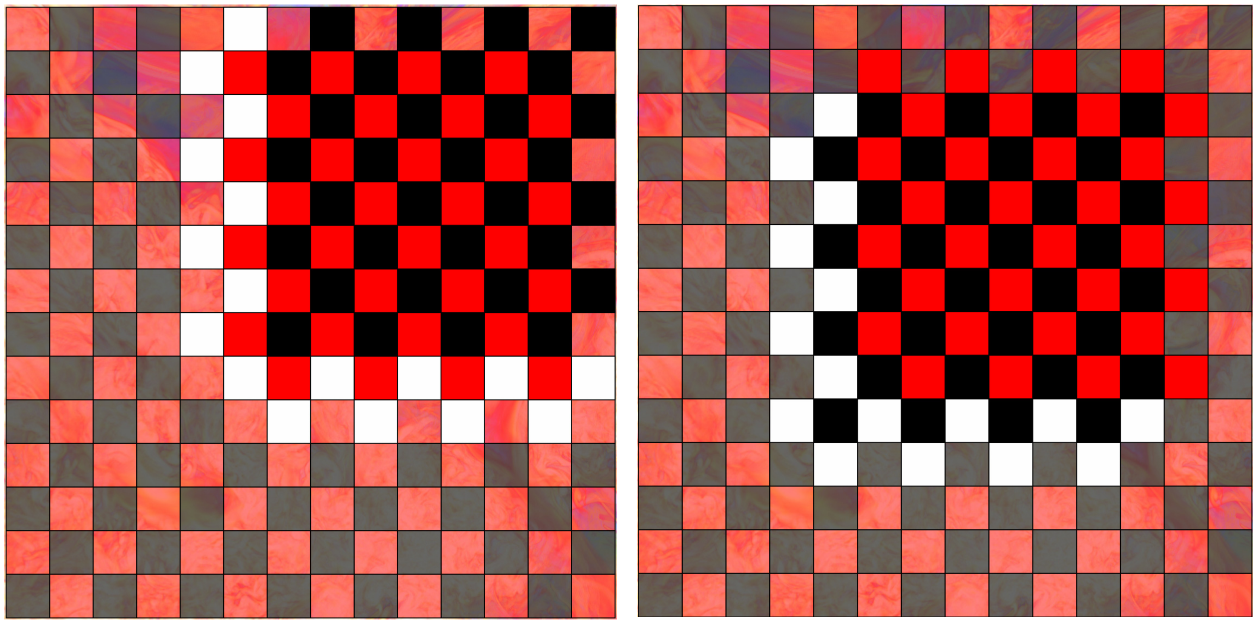


Figure 2. Illustration of the stages of the red-black iteration to be fused for efficient SPU vector computation.

In 2D, we need 4 flops to generate each new red- or black-cell value, while in 3D we need 6. In 2D, we organize the vertical columns of darkened black or red cells in the diagrams above into quadword data structures. Because we handle alternate columns of cells slightly differently, this gives us 4 quadwords of each type, so that all our operations will boil down to quadword operations each of which will be repeated 4 times. This is a kind of loop unrolling, and our experience with the Cell SPU emulator shows that it very nearly eliminates all processor stalls caused by data dependencies. If we carry out 4 full 2-D red-black iterations, once we get going we will produce 32 new black-cell values for every 32 old ones read in. We must also read in 64 source term values, of course. For each of these 32 ultimate values, we must perform 32 flops. Thus we perform 8 flops (in vector mode) per main memory word accessed. If the algorithm goes at the peak rate, then these 8 flops will take only 2 clock ticks on a single SPU. In this time interval, the Cell processor as a whole may read or write 4 words, so that each SPU can just access half of its needed word of main memory. We see that we might as well perform another 4 red-black iterations before writing our results back to main memory. If we do this, the extra work will come at essentially no cost in wall clock time. Now we will have 16 flops to perform per word of main memory access, and the processor should just have enough main memory bandwidth to support this computation at the rate of 4 flops/clock-tick/SPU. At 3.2 GHz, this would be 100 Gflop/s. Because we will have to perform various quadword realignment operations as the computation proceeds, we expect, based upon emulating other algorithms, that this performance level would drop to something closer to 66 Gflop/s, which would relieve the pressure on the main memory bandwidth slightly. Also, of course, we would need to actually want all 8 of these iterations in order to be fully pleased with this performance number. In 3D, the situation is similar, but harder to illustrate in diagrams. We do 3/2 as much arithmetic in 3D for every main memory word accessed, but we probably could still not get away with doing just 4 red-black iterations at a time without losing performance.

Any implementation of a numerical algorithm for Cell must be very careful to keep the data context for the computation in the SPU local store extremely small. For the 2-D algorithm, this presents no problem. We need to retain just one value for each column of our grid to provide for all the white-cell data in our diagrams as we move our block of red or black cells to the right along the present row of grid blocks and then along the row above it. We also need to save 8 additional values along the left-hand edge of the present grid block. For a grid subdomain of 128^2 cells, this is just 132 data values, plus 16 more ghost cell values for a total of 140. When we add in the 32 values in the grid block we are working on, we have 172 values, or 688 bytes. We need to do this for all 16 red or black subiterations, so that altogether we must save 10.8 KB. Adding in 4 grid blocks of source term values increases this total by 16 KB to give 26.8 KB. All these numbers will take up hardly any space in the SPU's 256 KB local store. In 3D, we will need to store much more data than this. The 3-D equivalents to the 2-D data values we have just enumerated are precisely 8 times greater, since our block of grid cells is now 8^3 rather than 8^2 in number. This data will then require 214 KB in the local store. However, in 3D we also need to store one value per grid line of grid cells in the new z-dimension, and we must do this for all of the 8 red- or black-cell intermediate quantities that we use in our calculation. Counting ghost cell values, this amounts to $18^2 \times 8^2 \times 8 \times 4 = 648$ KB. Since this amount of data will not fit into the SPU's local store, we will have to either reduce the size of the grid subdomain from 128^3 to 64^3 cells, or get the 8 SPUs of the Cell processor to share this data among them. We will also need to reduce the size of our 3-D grid block from 8^3 to 4^3 cells, which will still offer ample opportunity for vector loops over as many as 16 elements. Such modifications will come at some performance cost. The cost of reducing the size of the grid subdomain, however, is the easiest to estimate. Our grid subdomain will consist of 512 grid blocks plus 488 ghost grid blocks if we perform only 4 full red-black iterations. In each of the ghost grid blocks we will perform about half as much computation as in the real ones, and this computation will be redundant with that performed in updating other grid subdomains. Thus 244/(512+244), or nearly 1/3 of our effort will be simply parallel implementation overhead. If we get 8 SPUs to cooperate on a 128^3 grid subdomain, we can reduce this overhead to only 17%, but performance will suffer some reduction due to the need for much finer synchronization of the computations performed by these 8 SPUs. Our experience to date with the SPU emulator shows that the need to perform the many necessary quadword alignment operations that this red-black relaxation algorithm demands will tend to reduce our hoped-for computation rate of 100 Gflop/s toward 66 Gflop/s for a single Cell processor, and the above argument suggests that about a third of this performance would consist of parallel implementation overhead. If we perform only 4 iterations at once, main memory bandwidth limitations may halve even this lower performance estimate to 33 Gflop/s/Cell on this algorithm. Nevertheless, this performance would still be truly stunning in the context of present supercomputing platforms.

We can estimate the level of system service that such 33 Gflop/s Cell computation would demand for a 3-D red-black relaxation algorithm for the Poisson equation. We will assume that each of the 8 SPUs independently performs 4 full red-black iterations on a grid brick of 64^3 cells. Since only the black cells are ever read or written, this is really a brick of $64^2 \times 32$ cells. We perform 48 flops/cell on these 1.05 million cells in our 8 grid bricks, for 50.3 Mflops in total. At 33 Gflop/s, this takes only 1.53 msec. In this short time interval, we must stream $8 \times 80^2 \times 40 \times 3 \times 4$ bytes = 23.4 MB into the processor, and we must simultaneously stream $8 \times 64^2 \times 32 \times 4$ bytes = 4 MB out of the processor. Together, this requires a data streaming rate of 27.4 MB / 1.53 msec = 18.0 GB/sec. This is under the specified capability of 25.6 GB/sec, but not by very much. This is not surprising, since we halved our performance estimate above because of the limited main memory bandwidth. However, it is the network data streaming rate that would be most difficult to provide. Let us suppose that our 8 SPUs have in fact collaborated on the update of a single grid brick of 128^3 cells that has been decomposed into 8 sub-bricks. In this case, assuming that the neighbor brick records are maintained by other Cell processors on the network, we would need to both send and receive in our 1.53 msec the boundary data of $(144^2 \times 72 - 128^2 \times 64) \times 3 \times 4$ bytes = 5.09 MB (including source term values). This would require streaming 3.33 GB/sec continuously in each direction simultaneously, an extremely tall order. On any present equipment, this seems very unlikely to happen. Schwarz-type relaxation iterations, in which boundary data is held fixed while the interior domain is relaxed, would therefore seem more practical, even if they do converge more slowly. We need to get our data streaming rate over the network down by at least a factor of 8, so that we would then be able to perform 32 full red-black relaxation iterations between each data communication event. These extra iterations would not be useless, but they would not be as helpful as ones in which boundary data is exchanged more frequently. Of course, in the multigrid relaxation scheme, we would also have to communicate additional data, but this data refers to coarser grids, and is less voluminous.

The value of performing more red-black iterations at a time, because they essentially come for free, can be qualitatively assessed by playing with a multigrid relaxation program developed for a computational methods course at the University of Minnesota. This can be found online at www.lcse.umn.edu/-multigrid with associated documentation. This Fortran and Visual Basic program runs on Windows platforms. In its computational kernels, it embodies a Fortran code implementation that is vectorized but that is not as memory efficient as the one outlined above, and it is not as fast. Nevertheless, it allows the user to experiment with different multigrid strategies and, in particular, with different numbers of iterations performed on each grid at each stage.

Summary. The Cell processor offers a very large potential performance boost for scientific simulation codes. To realize this potential benefit, however, these codes must be restructured so that their local memory workspaces are made extremely small and their use of vector arithmetic is also enhanced. The second challenge is familiar and not at all hard by now, but the first challenge can be quite difficult. We have presented general techniques by means of three fairly generic examples. These techniques, which involve a massive fusion of what are normally programmed as distinct phases of computation or distinct subroutines containing many distinct loops, can be applied quite broadly. It can also be automated in a precompiler tool. The restructured and transformed code can be expressed clearly, although not easily, in standard Fortran from which the programmer's intent can be recognized by, at least, the Intel Fortran compiler. The necessary code restructuring and transformation for Cell proves to be worth the trouble even on standard CPUs, where in our experience with PPM it delivers a performance boost of from 50% to 100%, depending upon the particular CPU.

Acknowledgements. This work has been supported by DoE Office of Science grant DE-FG02-03ER25569 from the MICS program and by a contract from Los Alamos National Laboratory through the DoE ASC program. It is also a pleasure to acknowledge discussions during this work with Peter Hofstee, Karl-Heinz Winkler, Pen-Chung Yew, David Porter, Robert Lowrie, and Ben Bergen. Robert Lowrie, Ben Bergen, and Stephen Hodson are especially acknowledged for running our test programs on many occasions on real pre-production Cell hardware at Los Alamos.

References.

1. Woodward, P. R., and P. Colella, "High-Resolution Difference Schemes for Compressible Gas Dynamics," *Lecture Notes in Phys*. **141**, 434 (1981).

2. Woodward, P. R., and P. Colella, 1984. "The Numerical Solution of Two-Dimensional Fluid Flow with Strong Shocks," *J. Comput. Phys.* **54**, 115-173.

3. Colella, P., and P. R. Woodward, 1984. "The piecewise-parabolic method (PPM) for gas-dynamics simulations," *J. Comput. Phys.* **54**, 174-201.

4. Woodward, P. R., 1986. "Numerical Methods for Astrophysicists," in *Astrophysical Radiation Hydrodynamics*, eds. K.-H. Winkler and M. L. Norman, Reidel, pp. 245-326. Available at www.lcse.umn.edu/PPB.

5. Woodward, P. R., 2005. "A Complete Description of the PPM Compressible Gas Dynamics Scheme," LCSE internal report available from the main LCSE page at www.lcse.umn.edu. A shorter version of this paper is to appear in *Implicit Large-Eddy Simulation: Computing Turbulent Fluid Dynamics*, edited by F. Grinstein, L. Margolin, and W. Rider, Cambridge University Press.

6. Woodward, P. R., and S. E. Anderson, "Portable Petaflop/s Programming: Applying Distributed Computing Methodology to the Grid Within a Single Machine Room," Proc. of the 8th IEEE International Conference on High Performance Distributed Computing, Redondo Beach, Calif., Aug., 1999; available at www.lcse.umn.edu/HPDC8.

7. Woodward, P. R., Anderson, S. E., Porter, D. H., and Iyer, A., "Cluster Computing in the SHMOD Framework on the NSF TeraGrid," LCSE internal report, April, 2004, available on the Web at www.lcse.umn.edu/turb2048.

8. Woodward, P. R., and D. H. Porter, 2005. "PPM Code Kernel Performance," LCSE internal report available at www.lcse.umn.edu.

9. van Leer, B., 1977. "Towards the Ultimate Conservative Difference Scheme. IV. A New Approach to Numerical Convection," *J. Comput. Phys.* **23**, 276-299.

10. Woodward, P. R., 2006. "PPB, the Piecewise-Parabolic Boltzmann Scheme for Moment-Conserving Advection in 2 and 3 Dimensions," LCSE Internal Report available at www.lcse.umn.edu/PPB.