

Moving Scientific Codes to Multicore Microprocessor CPUs

A restructuring method for implementing numerical algorithms for scientific computing can help them run efficiently on the IBM Cell processor and other multicore CPUs. Using the PPM gas dynamics algorithm as an example, this work walks step by step through code transformation techniques that can boost the performance of scientific simulation codes.

The IBM Cell processor represents the first and most extreme of a new generation of multicore CPUs. For scientific codes that can be formulated in terms of vector computing concepts, as far as we know, the Cell is the most rewarding. In this article, our team at the University of Minnesota presents a method for implementing numerical algorithms for scientific computing so that they run efficiently on the Cell processor and other multicore CPUs. We present our method using the Piecewise-Parabolic Method (PPM) gas dynamics algorithm^{1,2} but believe that many other algorithms could benefit from our approach. Nevertheless, the code transformations are difficult to perform manually, so we're undertaking an effort to build simplified tools to assist in at least the most tedious of the code transformations involved.

Fitting the Cell Processor into the CPU Spectrum

From a programming perspective, the Cell proc-

essor seems to be midway between standard commodity CPUs and GPUs in programming difficulty (although we haven't attempted GPU programming as yet for our fluid codes). The Cell has eight cores, plus an additional core to manage these eight, whereas the Intel and AMD CPUs have only four cores and Nvidia GPUs have 128. Each worker core, or synergistic processing element (SPE), on the Cell has 256 Kbytes of on-chip memory, whereas each Intel or AMD core has 2 or 3 Mbytes and each Nvidia core has 1 Kbyte. Getting everything that should fit into this on-chip memory to actually fit there is the principal programming challenge. For the Cell, it's difficult, for Intel or AMD, much easier, and for Nvidia, it's simply out of the question for many numerical algorithms.

The capability of the Cell core (the SPE) is also midway between that of Intel or AMD and Nvidia. We can loosely characterize the capability differences: the Intel and AMD cores have all the standard features, and the Cell SPE has all that we might need to do "core" scientific computing (that is, computing as opposed to messaging, I/O, graphics, and so on). The Nvidia core, presumably, has all that we need to do graphics. The features left out of the Cell SPE, as well as the small size of its private cache, enable more of these cores to be accommodated on a single chip than would be possible for general-purpose CPU cores. This

1521-9615/08/\$25.00 © 2008 IEEE
COPUBLISHED BY THE IEEE CS AND THE AIP

PAUL R. WOODWARD, JAGAN JAYARAJ, PEI-HUNG LIN,
AND PEN-CHUNG YEW
University of Minnesota

provides an incentive to programmers to handle the extra difficulty in order to receive the benefit; if a scientific application doesn't need the features omitted in the Cell SPE, then their elimination in favor of more cores per chip is a good thing. For the fluid dynamics codes we've moved to the Cell over the past 18 months, this benefit has definitely been worth the programming cost.

The Cell processor also introduces two significant features that aren't directly accessible through high-level programs on Intel or AMD chips. The first is that the on-chip memory assigned to each SPE core is a local store rather than a cache. The second is that asynchronous direct memory access (DMA) can be invoked from user code. For scientific codes, both of these features give the programmer the benefit of control but at a new programming cost. We believe that this benefit is worth it. In fact, we argue that the programming cost of trying to imagine and account for what the caching and prefetching hardware on an Intel or AMD core might conceivably do in reaction to any given high-level code expression is far greater than the programming cost of the control delivered by the Cell's local store and its DMA instructions. And, of course, good automated code transformations can, in principle, reduce the programming burden of explicitly managing the local store.

The Challenge of Cell Programming

The 256-Kbyte local store of the Cell processor SPE core must accommodate both the program and the data workspace needed for the computation. Data can be streamed into the local store using the DMA capability; the program then operates on this data using the local store as a data workspace, and the results are streamed back to main memory using the DMA again. This data streaming can occur simultaneously with arithmetical computation. Because it costs just as much per byte to bring the program into the local store as it costs to bring in the data, it's best to amortize the cost of loading the program by having it operate on more data. This strongly encourages programmers to use the DMAs for data streaming and can be highly efficient if we can write the program so that a great deal of useful computation is performed on the data after it's brought in and before the ultimate computed results are streamed back out.

We can estimate how much work this entails for any given multicore CPU if we know its computing capability and its main memory bandwidth. The idea is to keep the floating-point units as busy as possible, which means that they should es-

entially never be waiting for data to arrive from main memory into the local store. The concept of "as busy as possible" is algorithm and CPU core dependent. For our PPM algorithm, we figure this out by executing the program on a tiny data set that fits entirely into the local store and determining the computation rate. For our various gas dynamics applications, it varies from one code to the next, but for our best performer (single-fluid PPM), it's roughly 7.7 Gflops per second per core (Gflop/s/core), which represents only approximately 30 percent of the peak performance.

With this, we can estimate the bandwidth to main memory that would be required to maintain this 30 percent of peak performance that we know is possible if all our data is on chip. Knowing the number of flops performed per grid cell—742 for single-fluid PPM and 1,161 for the two-fluid version—and the number of bytes read in and written back per grid cell—48 and 120, respectively—we can estimate the bandwidth to main memory required to keep this program running at this speed. The computational intensities of these two PPM code versions are 62 and 39 flops per word (flops/word), respectively. At 30 percent of peak performance, we execute 2.4 flops per clock cycle, so that we have 26 or 16 clock cycles in which to transfer the needed word. Given that the eight SPU cores on the Cell processor share an ability to stream four words per clock, each can stream only half a word per clock cycle. This means that we need, in principle, only two clock cycles to make our data exchanges with main memory, but we have 26 or 16 available to us. Indeed, when 16 SPEs in a dual-Cell blade cooperatively update a grid brick of 32^3 cells, the cost of the DMA operations appears to be negligible, and the performance is 7.7 Gflop/s/core. We therefore estimate that a numerical algorithm eight times less computationally intensive (at only 5 flops/word) might still achieve 30 percent of peak performance.

By artificially increasing the size of our data transmissions, we can test this hypothesis. Unfortunately, we find that computational intensities of 62, 31, 15.5, and 7.75 flops/word result in performance levels of 7.7, 7.2, 4.4, and 2.7 Gflop/s/SPE, respectively. We therefore conclude that as a rule of thumb, programmers should aim to achieve computational intensities of 20 or more flops/word. In special cases of interest to us, we find that the two-fluid PPM code, with 63 percent of the computational intensity, runs at 83 percent of the speed in Gflop/s of single-fluid PPM. Also, we find that performance of single-fluid PPM drops to 5.7 Gflop/s/core when we have 16 SPEs update

a grid brick of 128^3 cells. We speculate that this performance reduction is related to translation look-aside buffer (TLB) misses.

This discussion gives us a new reason for understanding old design trade-offs. It seems like common sense that the larger a program resident in the local store is allowed to be, the better its chance of being able to carry out the “great deal” of useful work that high performance requires. This introduces a trade-off between the local store’s size and either the difficulty of writing the program by restructuring it to satisfy this requirement for computational intensity or the main memory bandwidth required for the program to run efficiently without stalling for lack of data.

In this article, we set out programming techniques that work for our algorithms for present CPUs, including the Cell. Readers can judge whether the associated programming difficulty is worth the resulting performance benefits. We can reduce programming difficulty by automated code transformations, which we’re now working to implement.

You might think that to support a sufficiently intense computation from data residing in the local store it would be necessary to use most of this on-chip storage to hold data rather than the program. But in our experience, this turns out not to be the case. Instead, we find that the program requires a significant fraction of on-chip storage for two reasons. First, for our codes, it’s necessary to encapsulate an enormous amount of computation into the program in the local store to achieve the needed computational intensity. Second, this code’s performance is enhanced by approximately 16 percent via unrolling the vectorizable loops (some loops benefit more than others). This loop unrolling produces a great deal of additional code. In our codes, which vectorize the computation over grid planes of 16 grid cells each, we observe that the code roughly doubles in size if all the loops are completely unrolled, which does produce the best performance. The program’s swelling through loop unrolling and the need for computational intensity puts pressure on the utilization of the local store’s remaining space for the data workspace.

We address this issue using code transformations that dramatically reduce the workspace size through efficient reuse of temporary storage. We also keep our vector lengths short, at 16 elements. The same code using vectors of length 36 is approximately 5 percent slower, for unknown reasons, and hence we conclude that 16 is a good number. For performance, the vector length

must be an integer multiple of four because the CPU core doesn’t actually perform true vector arithmetic, but instead pipelined four-way SIMD arithmetic. Reducing the vector length from 16 to 12, eight, or even four will reduce the data workspace’s size in the local store by a corresponding factor, but we can’t reduce below a vector length of four without a significant sacrifice of performance. Therefore, we can shrink the data workspace only so far.

We can get an idea of code and data workspace sizes from the examples of our two PPM versions. For single- and two-fluid PPM, we do 742 and 1,161 flops per grid cell, respectively. The computational portions of these programs that must fit into the local store boil down to roughly 1,500 and 3,200 lines of Fortran, after applying the transformations we describe in this article. These programs take up 43 and 97 Kbytes in the local store, while their workspaces take up 39 and 76 Kbytes. Thus, we can conclude that a program twice the length of two-fluid PPM will have trouble fitting into the local store along with its data workspace. In fact, an earlier version of this program that updated grid briquettes of 216 cells each rather than groups of four briquettes of eight cells each demanded approximately 5 Kbytes more space than the local store of the Cell processor SPE allows.

Our experience indicates that fitting an entire multiphysics code and its associated data workspace into the local store of the Cell processor SPE is unlikely to be workable. Instead, we can apply the physics operators in succession, using the operator splitting technique that is common for such codes. Another option is to spread the code over multiple SPE cores on the same CPU chip. In such a strategy, we should be able to spread the workspace over these multiple SPE cores as well. We will show that once the code is transformed for execution in the SPE, it’s already fully pipelined. Dividing it into segments and setting up a pipeline of multiple SPEs to do this processing then isn’t as hard as it would otherwise be. Nevertheless, such an implementation would be considerably more difficult than those we discuss in this article. There’s also the need to balance the loads on the individual cores, which could be difficult for some algorithms. This approach would, however, allow the program to swell by a factor of eight without necessarily reducing the level of aggregate performance on the Cell processor.

Transforming Simple Fortran to Fast Fortran Code Expressions

In our work with multicore processors, and par-

ticularly with the Cell, we've used a set of general code transformations, which build on our earlier work.^{3,4} We performed these transformations manually, and we're presently constructing tools to perform them automatically. Our tools will assume that the target programmer is highly expert and willing to communicate to our tools (via directives) with special knowledge about the program. Our tools will also assume that in exchange for the benefit of the automated code transformations, such a programmer will be willing to fairly strongly restrict the range of standard Fortran code expressions used. Thus, our tools will be much easier to build than standard industry tools. This feature will let us get our tools into the hands of other scientists much earlier.

This section lays out the general techniques that we've used and that we'll soon automate. Some of these techniques are available in one form or another in compilers today, but compilation of our codes in the standard fashion reveals that the performance benefits we've obtained can't be achieved through simple compilation. The code must be transformed first and then compiled. A possible reason for this is the extensive nature of the code transformations we use. Another reason is that for these transformations to deliver their full benefits, the programmer must restructure the fundamental layout of data in the main memory, a transformation that's unlikely ever to be carried out by a compiler except at an expert programmer's explicit request, through directives.

The code restructuring and transformation techniques we describe here are general, but we don't expect them to apply to all scientific codes; at this time, they're domain-specific techniques. As we gain familiarity with them, we'll understand better how broad a domain they can apply to. Other efforts using different techniques are available on the Los Alamos National Laboratory's Roadrunner Web page (www.lanl.gov/orgs/hpc/roadrunner) and elsewhere.⁵ Our team has also provided Roadrunner tutorial materials on this site that show a comparison of running our PPM code with 32- and 64-bit arithmetic. After carefully revising this code to avoid needlessly large rounding errors, we find that 32-bit arithmetic is entirely adequate for the fluid-flow problems our code addresses, and it delivers roughly double the computation rate on most processors.

The Grid Briquette

The four-way SIMD processing that's the only 32-bit floating-point mode offered by the Cell processor SPE demands that we provide a cer-

tain level of computation uniformity if we want the SPE to perform. For fluid dynamics codes like ours, this translates into a demand for logical grid uniformity on some level that we'll call a *grid briquette*. Think of this briquette, which we force to be a cube, as a sugar cube—it's an easy matter to construct physical models representing our operations using sugar cubes (see Figure 1).

To each grid briquette we assign a small, contiguous record in main memory. This record contains the values of all the fluid dynamical state variables in the briquette's grid cells. The grid cubes in Figure 1 display the geometrical layout for the computation and the data layout in memory. DMAs transfer groups of grid briquettes four at a time to and from the SPU local store (see the arrows in the figure). The SPU then extracts single grid planes of data from these and places them into the data workspace, indicated by the grid block at the lower right. Thus, the large grid brick at the left in the diagram is in the main memory, whereas the duplicate briquettes and data workspace, holding just five planes of grid cells, at the right are in the processor's on-chip local store or cache memory. In the very small data workspace, 742 flops per grid cell are performed in pipelined, four-way SIMD mode at 5.7 Gflop/s to update one grid plane of 16 cells. The updated plane of cells (in red) is then transferred, and potentially transposed, into the group of 2^2 grid briquettes being prepared for reinsertion into the grid brick data array in the main memory. The dashed grid briquettes on the left in the figure, which are required to update the briquettes shaded in red, are contained in a message-passing interface (MPI) message received from another node of a machine's interconnect network.

Code Preprocessing

Because the SPE processes four words simultaneously, our code transformations will demand that this grid briquette be no smaller than 2^3 grid cells. We've experimented with codes using briquettes of 2^3 , 4^3 , and 6^3 cells and have settled on 2^3 . This smallest possible briquette is the most flexible; it also permits both the code and the data workspace to be as small as possible. By processing multiple briquettes simultaneously, it lets us make the code and workspace fit optimally into the SPE's local store. A fundamental feature of the codes we'll write is that each grid briquette has a *record*—that is, a sequential set of words—in main memory that fully describes it for the computation's purpose. If we decide to process a briquette, we need only read this single record by means of a single,

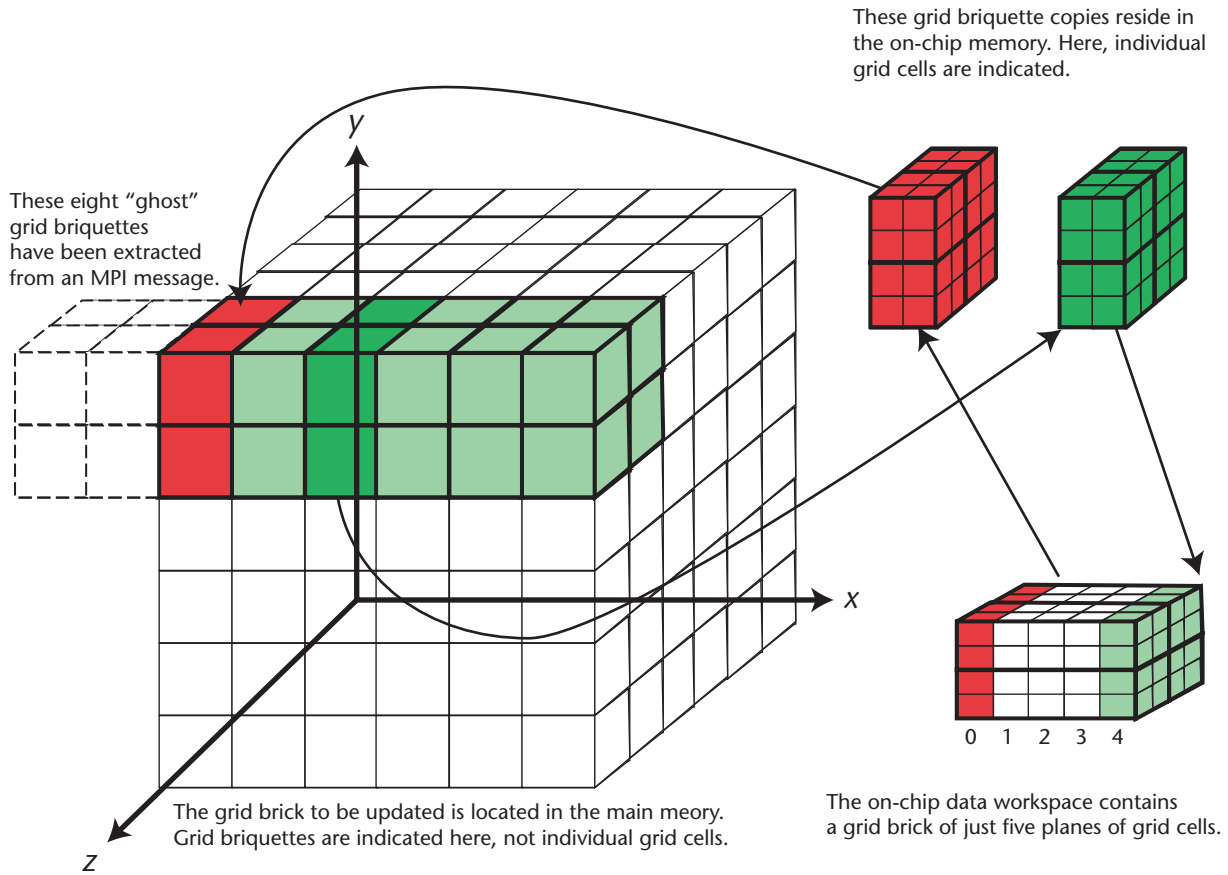


Figure 1. The code execution for the Cell processor SPU. A grid brick of 6^3 grid briquettes of 2^3 cells each is on the left. The shaded grid pencil of $2^2 \times 6$ briquettes will be updated for the x-pass of the PPM algorithm as a result of a single call to the Fortran subroutine encapsulating the SPU program. The dashed lines on the left indicate 2^3 ghost briquettes, which are extracted from a separate, contiguous data structure in main memory.

serial data transfer. For the SPE, this is a DMA, and for any other processor, it's a single-array assignment loop copying the record in main memory into a locally dimensioned array on the stack that's (presumably) cache resident. For single-fluid PPM, which updates six variables per grid cell, the data record corresponding to the grid briquette is 192 bytes, which is somewhat inefficiently small. However, for this code, we can read these records in eight at a time if we pack them appropriately. For two-fluid PPM, which updates 15 variables per grid cell, the data record is 480 bytes.

We pack data records together in main memory in standard Fortran 3D array order to form a data brick corresponding to a grid brick. This is a hierarchical data structure:

```
dimension DD(2,2,2,6,nbx,nby,nbz)
```

Here, the data is organized into a 3D brick of

48-word briquette records, each of which is the concatenation in memory of briquette records for six fluid-state variables. The code module that executes in the Cell SPE processes a sequence of these grid briquettes arranged in a line in the direction of a 1D pass of the algorithm. Actually, this code module processes four such lines of briquettes simultaneously, making up a grid pencil of $4 \times 4 \times 2m$ grid cells, as Figure 1 shows. The code that does this can be produced by a series of transformations that start from a simplified Fortran (SF) expression of the numerical algorithm acting on just a single grid briquette. We can easily debug this SF code by writing it with the briquette size as a parameter, which we can set to, say, 128 for the purpose of testing. Once the code is debugged and then transformed into the much faster Fortran expression, we can reset this parameter to two.

Our codes consist of 1D passes. In each pass,

only derivatives in the direction of the pass are accounted for. We can exploit this fact to produce arithmetic for which all operands will be aligned on four word boundaries, which is the most computationally efficient. Such a 1D pass requires ghost cell data only on the ends of the grid pencil that will be updated. For single-fluid PPM, to update a given grid cell, information from four cells on either side of it in the direction of the pass is required. For two-fluid PPM, instead of four ghost cells on each side, we need five. If we write code to update a single briquette, we need to dimension arrays for our fundamental fluid-state variables, as in the following pressure array:

```
dimension p(n,n,1-nbdy:n+nbdy)
```

Here, we've demanded that our briquette be a cube, and the parameter `nbdy` assumes the value four for the single-fluid code and five for the two-fluid code. None of the temporary arrays in our code will need to be larger than this, so for simplicity, we make all temporary arrays have precisely this same shape.

We can now express the numerical algorithm, assuming that all subroutines have been in-lined (easily done automatically) as a series of triple loop nests. In each such loop nest, the computation extends over the entire grid briquette plus some portion of the ghost cell regions. We can put into a single loop nest all computations that need to be carried out over the same extents in the third, slow-running index. It will then generally be possible to order the loop nests so that these extents grow progressively smaller from one loop nest to the next. The reason for this is simple: if a given expression involves a particular quantity at more than one value of the third index, then this expression can't be evaluated in the loop nest in which that particular quantity is evaluated. It must instead be placed in a loop nest with a smaller extent in the third index.

It's perhaps easiest to see this through examples taken from the single-fluid PPM code:

```
do i = 1-nbdy,n+nbdy
do k = 1,n
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
do j = 1,n
pv = p(j,k,i) / rho(j,k,i)
ceulsq = gamma * pv
ceul2i(j,k,i) = 1. / ceulsq
ceul = 1. / sqrt(ceul2i(j,k,i))
c(j,k,i) = ceul * rho(j,k,i)
```

```
. . .
enddo
enddo
enddo
```

In this loop nest, we compute the inverse square of the Eulerian sound speed, *ceul2i*, and the Lagrangian sound speed *c*, given the pressure *p* and density *rho*. We can evaluate these quantities in all the cells of our brick and in all the ghost cells. In the next loop, we take values of *c* computed here in two adjacent planes of cells and form their average, *cl*, which is an estimate of the Lagrangian sound speed at the left-hand interface of the right-most of these two planes of cells (hence the suffix *l*). We then use this result in the evaluation of an estimate of the Riemann invariant differences, *drplsl*, between the two adjacent grid planes in the direction of the pass:

```
do i = 2-nbdy,n+nbdy
do k = 1,n
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
do j = 1,n
cl(j,k,i) = (c(j,k,i-1) +
c(j,k,i)) * .5
clinv(j,k,i) = 1. / c(j,k,i)
temp = (p(j,k,i) - p(j,k,i-1)) *
clinv(j,k,i)
drplsl(j,k,i) = (ux(j,k,i) -
ux(j,k,i-1)) + temp
. . .
enddo
enddo
enddo
```

We couldn't place this computation of *cl* in the previous loop nest because both of the referenced values of the sound speed *c* aren't available there. In the next loop nest, we form various cell-centered quantities using cell interface quantities such as *drplsl*:

```
do i = 2-nbdy,n+nbdy-1
do k = 1,n
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
do j = 1,n
dasppm(j,k,i) = (drplsl(j,k,i) +
drplsl(j,k,i+1)) * .5
a6sppm(j,k,i) = (drplsl(j,k,i) -
drplsl(j,k,i+1)) * .5
. . .
enddo
```

```

        enddo
    enddo

```

The basic strategy here is to organize the computation so that we do all the work that needs to be done over the full extent in grid planes (index i) in the first loop nest. Then, we do all the work we need over this extent less just one plane in the second loop nest, and so on. In the final loop nest (the ninth one for PPM), we at last evaluate the new, updated cell averages over the extent of our grid brick but with no ghost grid planes included. Such an organization of the computation is always possible. The boundary conditions for the problem tell us how to set the values in the ghost cells at the points in the code in which these ghost cell briquettes are read.

This coding style is easy to write, modify, and debug. Having all intermediate quantities computed over the entire grid brick—which for debugging, is the entire grid—is tremendously convenient. We can insert after any loop nest a call to a routine that will produce a 3D brick of bytes for any such quantity, which lets us visualize it interactively with a volume rendering tool such as the Laboratory for Computational Science and Engineering’s hierarchical volume renderer (HVR) utility (www.lcse.umn.edu/hvr/hvr.html). SF’s only disadvantage is its performance, which we can fix by applying the sequence of code transformations.

Code Transformations

We can take our code in this form, with parameter n giving the size of the grid brick set to 128 or 256 for the purpose of testing, and simply compile it using Intel’s Fortran compiler, version 9.1 or earlier. This compiler will respect the vectorization and data-alignment assertions we’ve made (version 10, apparently, won’t do this) and produce reasonable but slow code that’s suitable for debugging and testing. To speed this code up, we must do a series of transformations.

First, we fuse all the loops over j and k so that we have loops over a single index jk that runs from 1 to n^2 . All our arrays collapse so that they have only two rather than three indices. We can do this with equivalence statements or simply eliminate the three-index forms. The resulting code expression is less clear, but it will be more efficient. Now we right-justify all the outer loops over the index i . We rewrite all these loops, if necessary, so that the extent’s upper limit is always $n + nbdy$, the upper limit of the first loop nest. Only the third of these three loops will change. It becomes

```

        do i = 3-nbdy,n+nbdy
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
        do jk = 1,n*n
            dasppm(jk,i-1) = (drpls1(jk,i-1) +
drpls1(jk,i)) * .5
            a6sppm(jk,i-1) = (drpls1(jk,i-1)
- drpls1(jk,i)) * .5
            . . .
        enddo
    enddo

```

We now fuse all the outer loops over i to produce a single such loop running from $1-nbdy$ to $n+nbdy$:

```

        do 9000 i = 1-nbdy,n+nbdy
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
        do jk = 1,n*n
            pv = p(jk,i) / rho(jk,i)
            ceulsq = gamma * pv
            ceul2i(jk,i) = 1. / ceulsq
            ceul = 1. / sqrt(ceul2i(jk,i))
            c(jk,i) = ceul * rho(jk,i)
            . . .
        enddo
        if (i .lt. 2-nbdy) go to 9000
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
        do jk = 1,n*n
            cl(jk,i) = (c(jk,i-1) + c(jk,i))
                * .5
            clinv(jk,i) = 1. / c(jk,i)
            temp = (p(jk,i) - p(jk,i-1)) *
                clinv(jk,i)
            drpls1(jk,i) = (ux(jk,i) -
                ux(jk,i-1)) + temp
            . . .
        enddo
        if (i .lt. 3-nbdy) go to 9000
!DEC$ VECTOR ALWAYS
!DEC$ VECTOR ALIGNED
        do jk = 1,n*n
            dasppm(jk,i-1) = (drpls1(jk,i-1) +
drpls1(jk,i)) * .5
            a6sppm(jk,i-1) = (drpls1(jk,i-1)
- drpls1(jk,i)) * .5
            . . .
        enddo
        if (i .lt. 4-nbdy) go to 9000
        . . .
    9000 continue

```

This form of the code is much more efficient,

but we will improve it further. In this formulation, the code is pipelined so that it traverses the grid brick only once. Grid planes of temporary data values are produced and then almost immediately consumed, so that for each temporary array, such as *drplsl*, only a small number of grid planes are “live” in that they’re accessed in a single traversal of the entire outer loop on *i*. In fact, in the full PPM algorithm, *drplsl* appears in four loops after the last one in the last code example, and grid planes *i*, *i* – 1, and *i* – 2 appear in the fused outer loop’s transformed code. This means that instead of allotting space for an entire grid brick plus ghost cells for the temporary variable, we need only allot space for three grid planes. We can assign three integer variables, *i2m0*, *i2m1*, and *i2m2* to designate planes *i*, *i* – 1, and *i* – 2, respectively. At the outset, we can assign values 0, 1, and 2 to these integer variables. Then, at the beginning of each iteration of the outer loop over *i*, we perform a barrel shift of these integer values. If we redimension *drplsl* via

```
dimension drplsl(n*n,0:2)
```

then the three planes of data in *drplsl* will be constantly reused. In Figure 1, where the on-chip data workspace is indicated by the five-plane grid brick at the lower right, these three grid planes are numbered 4, 3, and 2, corresponding to indices in the fused loop of *i*, *i* – 1, and *i* – 2. For the Cell processor SPE, where we have a local store, we’ve clearly saved an enormous amount of space by reducing the array *drplsl* to just three grid planes. Additionally, no transfers of this array’s planes to or from the main memory will be necessary at all. For a processor core with a cache memory, the same result should also occur, in effect, because the values in this tiny array will never be written back to main memory until the end of the entire computation, after which these three planes of data can have been overwritten in the cache literally millions of times.

This transformation results in a huge compression of the data workspace in the on-chip memory, be it a local store or cache, which has a tremendous impact on performance, roughly tripling or even quadrupling it on most processor cores if the parameter *n* is small enough. Although this is wonderful, it unfortunately comes at an enormous cost in code readability and maintainability. Each temporary array will have a potentially different set of active grid planes, and it becomes extremely difficult to determine code correctness by inspecting the vast array of index variables.

Bugs can take a week or more to locate, and writing correct code on the first try becomes a rare occurrence. This is an unacceptable situation for codes of any significant complexity that we ever expect to modify.

There’s only one reasonable solution, short of accepting SF’s performance rather than the faster Fortran code expression: an automatic code transformation tool such as the one we’re now building. Such a tool could, of course, be a compiler, but we’ve found no evidence that any existing compiler includes such transformations, at least ones that go through all the remaining steps (which we set out next) necessary to achieve the code’s full potential on the hardware.

Briquette Updating

At this point, we have the entire algorithm fully pipelined and the on-chip data workspace greatly reduced in size. It’s now possible to inspect the code and identify temporary single-grid-plane arrays that can be equivalenced together without

We set up our pipeline at some computational cost but then reap the benefit by updating any additional briquettes at peak efficiency.

affecting code correctness. This will collapse the data workspace still further but will make the code extremely confusing and/or impenetrable to read and understand. This is of no concern if this transformation is performed by an automated tool that has been fully debugged—the resulting code is handed to a standard compiler without inspection, and the compiled code executed. But this code is incomplete because it updates only a single grid briquette. We must add an outer loop over briquettes.

A key point here is that the pipelining we’ve performed can be exploited to make updating several briquettes in a row in the direction of the 1D pass more efficient than updating each briquette separately in series. In updating the first briquettes, we set up our pipeline at some computational cost. Then, we reap the benefit by updating any additional briquettes at peak efficiency. For single-fluid PPM, we must fully process four briquettes before our pipeline is completely set up. During this processing, no results are generated that are suitable to write back to main memory. For each additional briquette we put into our pipe, we get out a fully updated briquette from

the other end. To update each grid cell, we must perform 742 flops, but before any grid cells are updated, we must perform 1,998 flops to establish the pipeline.

For the calculation to be efficient, we must amortize this cost of 1,998 flops over the cost of many updated cells. If we process four ghost cell briquettes and update eight “real” briquettes, then this overhead of setting up the pipeline is 14.4 percent of the total work. If we update 16 real briquettes, the overhead is only 7.9 percent of the total, and the computation is thus 92 percent efficient in this sense.

Our outer loop over grid briquettes should express our desire to prefetch the next briquette while we update the present one and to write back the previous new briquette while we fill in the values of the present new one. This is easily expressed using the Cell SPE’s DMAs. For other processor cores, we can write these fetch and store operations as array assignment loops, know that a compiler can in principle discover them to be streamable, and pray. It’s conceivable that our prayers will be answered, but not guaranteed. Evidence from our experience indicates that a benign agency outside of our control is somehow operating.

For the Cell processor, the code we’ve described so far is translated from its fast, transformed Fortran expression into C plus SPE intrinsics using a combination of an automatic translator that we wrote and a small amount of hand translation. We don’t do this translation often, choosing instead to debug our code on standard CPUs and perform the translation for the SPE infrequently. As our translation tool and the Cell software tools evolve, we’ll be able to change this style of work. We load this code into the SPE’s local store at the outset of the computation, and we never move it. To update the entire grid brick, which is a 3D array of briquettes, as Figure 1 shows, we need to call this SPE program many times. This we coordinate from the PowerPC processor element (PPE), using mailbox messages. When we have all 16 SPEs in a dual-Cell blade cooperatively update a single grid brick in this fashion from its shared memory, the delivered performance is 91.2 Gflop/s, or 5.7 Gflop/s/core. In our Fortran code, the portion that we assign to the SPE consists of a single subroutine. In the SF expression, we might have many subroutines called from this routine, but in producing the faster pipelined Fortran expression, all these routines are in-lined.

Our codes perform 1D passes in symmetrized sequences of $xyzyx$. Therefore, in four of the six passes of such a sequence, we have the SPE trans-


pose the contents of each grid briquette record while it’s in the local store and before writing it back to main memory. Consequently, the contents of each grid briquette record are in the most useful order for fast, immediate processing when we copy them into the local store to perform the next 1D pass. This makes the calculation efficient. Each briquette record is transposed internally, but the locations of the briquette records in memory are unchanged. This optimization makes the indexing in the code confusing, and debugging wasn’t easy.

On top of this complexity, the SPE code separately writes copies of certain output briquettes to records within separate arrays in main memory that, upon completion, are transmitted to other network nodes to provide ghost cell data for other grid bricks. The formation of these message arrays is highly efficient, and their transmission while the computation is proceeding doesn’t slow the computation down noticeably. The high performance of this transformed code running on modern processors such as Cell puts significant stress on a large machine’s interconnect, but our experience to date indicates that present network technology can meet this requirement, so that the CPUs essentially never wait on data. We’ve seen that a grid pencil only 32 cells long can be updated in our fashion at 92 percent parallel efficiency (only 8 percent redundant computation in ghost cell regions), which means that there is enough work in a grid brick of only 32^3 cells to keep 16 cores busy and efficient. This granularity of parallel computation is so small that the network communication requirements for both bandwidth and latency are significant. These diminish by a factor of two for every eight-fold increase in the grid brick’s size. Taking the smallest granularity, with a million processor cores, updating two grid bricks of 32^3 cells by each team of 16 cores, we would have a global grid of 128,000 such bricks, or only $1,280^2 \times 2,560$ cells. For such a computing system, this grid would be modest indeed. The computing speed would be so great that we could perform 40,000 time steps, which should be enough for most problems, in roughly $(1,998 + 32 \times 742) \times 4^2 \times 8 \times 120,000 / (5 \times 10^9) = 79$ seconds.

To achieve this computation rate, however, the machine’s interconnect would need to enable two messages of 96 Kbytes each to arrive at and leave from each 16-CPU-core node every 0.33 msec. This would require a sustained bidirectional bandwidth to each node of 563 Mbytes/sec. We plan to exploit this scaling property of our restructured and transformed PPM code in the coming

months on the new Los Alamos Roadrunner system, with its 12,960 Cell processors (www.lanl.gov/orgs/hpc/roadrunner).

We found early on that our codes, after transformation into the fast, pipelined Fortran form and parallelized for large-scale execution, run so fast that problems are soon over, even when we use only one or two thousand cores. The time for our long-established style of data postprocessing then becomes for the first time in many years a great deal longer than the time to run the simulation.

We've been addressing this new feature of multicore computation in multiple ways. First, we introduced into the simulation code, in-lined in the SPE code module, the preprocessing of the computed briquette data to produce highly compressed data for output from the machine. This extra code amounted to roughly 1,000 lines of Fortran. It only rarely executes, so its cost as a fraction of the overall computing time is negligible. This data is streamed out of the machine, with one stream for every 64 worker cores. We've been pipelining the processing of this data in our lab at the University of Minnesota, the LCSE, through a string of earlier utility programs coordinated by a program running on a laptop or wireless device under the user's interactive control. The result is interactive supercomputing.^{6,7} It's still a bit crude, but we're working steadily to improve it. 

Acknowledgments

This work was supported by US Department of Energy (DoE) Office of Science grant DE-FG0203ER25569 from the Mathematical, Informational, and Computational Sciences (MICS) program and by a contract from Los Alamos National Laboratory through the DoE Advanced Simulation & Computing (ASC) program. We appreciate discussions during this work with Peter Hofstee, Karl-Heinz Winkler, David Porter, Robert Lowrie, and Ben Bergen. We thank Ben Bergen and Stephen Hodson for helping us run our test programs on many occasions on preproduction Cell hardware at Los Alamos during Fall 2006. We also acknowledge the donation to our lab, the LCSE, of two dual-Cell blades from IBM in January 2007 and the support for continued work with multicore CPUs from US National Science Foundation grant CNS-0708822. We also gratefully acknowledge support from the Minnesota Supercomputer Institute and, earlier, from the Pittsburgh Supercomputer Center for our work on interactive supercomputing.

References

1. P.R. Woodward, *A Complete Description of the PPM Compressible Gas Dynamics Scheme*, tech. report, Laboratory for Computational Science and Eng., Univ. of Minnesota, 2005; www.lcse.umn.edu.
2. F. Grinstein, L. Margolin, and W. Rider, eds., *Implicit Large-Eddy Simulation: Computing Turbulent Fluid Dynamics*, Cambridge Univ. Press, 2006.
3. P.R. Woodward and S.E. Anderson, "Portable Petaflop/s Programming: Applying Distributed Computing Methodology to the Grid Within a Single Machine Room," *Proc. 8th IEEE Int'l Conf. High-Performance Distributed Computing*, 1999; www.lcse.umn.edu/HPDC8.
4. P.R. Woodward and D.H. Porter, *PPM Code Kernel Performance*, tech. report, Laboratory for Computational Science and Eng., Univ. of Minnesota, 2005; www.lcse.umn.edu.
5. S. Williams et al., "The Potential of the Cell Processor for Scientific Computing," *Proc. 3rd Conf. Computing Frontiers*, ACM Press, 2006, pp. 9–20.
6. D.H. Porter et al., "Bursts of Stellar Turbulence," *Proc. Projects in Scientific Computing 2007*, Pittsburgh Supercomputing Center, 2007, pp. 34–37; www.psc.edu/science/2007.
7. P.R. Woodward et al., "Interactive Volume Visualization of Fluid Flow Simulation Data," *Applied Parallel Computing, State of the Art in Scientific Computing, Proc. PARA 2006 Conf.*, LNCS 4699, Springer, 2007, pp. 659–664; www.lcse.umn.edu/para06.

Paul R. Woodward is a professor of astronomy at the University of Minnesota, where he directs the Laboratory for Computational Science & Engineering (LCSE). His research interests include computational fluid dynamics in astrophysics, especially in applications to stellar evolution theory where 3D effects play a major role. Woodward has a PhD in physics from the University of California, Berkeley. He is a member of the IEEE and SIAM. In 1999, he also received the Gordon Bell Prize in the performance category with several collaborators. Contact him at paul@lcse.umn.edu.

Jagan Jayaraj is a research assistant at the University of Minnesota. His research interests include high-performance computing and compiler technology. Jayaraj has an MS in computer science from the University of Minnesota. Contact him at jaganj@lcse.umn.edu.

Pei-Hung Lin is a research assistant at the University of Minnesota. His research interests include high-performance computing and compiler technology. Lin has an MS in computer science from the University of Minnesota. Contact him at phlin@cs.umn.edu.

Pen-Chung Yew is a professor of computer science and engineering at the University of Minnesota. His research interests include architectural designs and compilation techniques for multicore processors. Yew has a PhD in computer science from the University of Illinois, Urbana-Champaign. He is a fellow of the IEEE. Contact him at yew@cs.umn.edu.

This article was featured in

computing|now

ACCESS | DISCOVER | ENGAGE

For access to more content from the IEEE Computer Society,
see computingnow.computer.org.



IEEE  computer society

Top articles, podcasts, and more.



computingnow.computer.org