Title: Simulating Rayleigh-Taylor (RT) instability using PPM hydrodynamics @Scale on Roadrunner (U)

Author(s): Paul Woodward, P.-H. Lin, G.M. Rockefeller, C.L. Fryer, G. Dimonte, W. Dai, R.J. Kares

Intended for: DECDC2010 Proceedings
Simulating Rayleigh-Taylor (RT) instability using PPM hydrodynamics @Scale on Roadrunner (U)

P. R. Woodward,*† J. Jayaraj,* P.-H. Lin,* G.M. Rockefeller,†† C.L. Fryer,††
Guy Dimonte,†† W. Dai,†† R.J. Kares,††

*University of Minnesota, Minneapolis, Minnesota 55455;
††Los Alamos National Laboratory, Los Alamos, NM 87545

The effect of initial conditions on the self-similar growth of the RT instability is investigated using a hydrodynamics code based on the piecewise-parabolic-method (PPM). The PPM code was converted to the hybrid architecture of Roadrunner in order to perform the simulations at extremely high speed and spatial resolution. This paper describes the code conversion to the Cell processor, the scaling studies to 12 CU’s on Roadrunner and results on the dependence of the RT growth rate on initial conditions. The relevance of the Roadrunner implementation of this PPM code to other existing and anticipated computer architectures is also discussed. (U)

Background and Motivation

The effect of initial conditions on the self-similar growth of the RT instability is investigated using a hydrodynamics code based on the piecewise-parabolic method (PPM). According to the self-similar model of Dimonte [1] and simulations by Ramaprabhu et al.[2], the multi-mode RT growth rate depends on the spectrum and amplitude of the initial perturbations. Narrowband initial perturbations grow by mode-coupling at a slow rate independent of the initial amplitude. Broadband initial perturbations grow by mode-competition at a rate that increases logarithmically with initial amplitude. To test these hypotheses more stringently, we exploit first-principles multi-fluid numerical calculations and the power of the Roadrunner machine. We combine the PPM gas dynamics scheme with a new multi-fluid treatment based upon the PPB moment-conserving advection scheme, as described elsewhere [3-5]. In this paper we describe the aspects of the code implementation that were specifically driven by the Roadrunner architecture and explain why these features are helpful on other existing machines as well as those anticipated in the near future. We also discuss how we have combined the PPM and PPB numerical schemes, and we discuss the resulting behavior of the code in simulating the behavior of unstable multi-fluid interfaces.

Hierarchical Code/Data Structure

Roadrunner has a hierarchical structure that is reflected in our PPM code. A three-level hierarchy of (1) cores on a single CPU, (2) CPUs at a single network node, and (3) nodes in a single machine is already standard today. Roadrunner complicates this structure by adding one specialized management core per CPU and another general purpose host core per CPU. There are also special I/O nodes on the machine, but the user is not allowed to run processes explicitly there. We exploit the heterogeneous nature of these processing elements on Roadrunner by having a single program in which a given process or thread discovers early on what type of entity it is and then executes the appropriate portion of the program. On other machines, we have found it useful to maintain this task differentiation, mapping tasks for the host processor cores to OpenMP master threads and retaining the special I/O processes but running them on...
The grid brick updated by a single CPU's multiple cores (lower left) and the extracted briquettes stored in cache.

unspecialized nodes of the machine.

The hierarchical decomposition of the work to match the structure of the machine is matched by a hierarchical structure of the problem data, as shown in Figure 1. At the left in the figure, a single grid brick is shown. The grid brick is the region of the physical domain that is updated by a single multicore (Cell) processor. It consists of a single data structure in the main memory of a node, which on Roadrunner means in the main memory of a Cell processor blade. It is augmented by associated MPI message buffers, from which ghost cell data (the dotted cubes at the far left in Fig. 1) is taken. The entire problem domain is decomposed into a large number of such grid bricks, grouped in "teams," which are described below. Each grid brick is built from a 3-D array of grid briquettes (the cubes in the grid brick at the left in Fig. 1). Each briquette has a contiguous data record in the main memory of a node that is either read or written all at once, for efficiency. The 15 variables that determine the multifluid state in a grid cell are packed together into the grid briquette record. Thus we have a briquette of density values, a briquette of pressure values, etc., concatenated to form the briquette data record. As shown at the right in Fig. 1, each briquette consists of a cube 2 grid cells on a side. With 15 variables and 32-bit precision, each briquette record is therefore 480 bytes. This is just enough to be read efficiently. The idea is that grid briquettes are read and written by individual CPU cores. Only one core may update a given briquette. Briquettes therefore serve in this code as quanta of data. This guarantees that reading and writing of data by CPU cores is highly efficient.

It is important to note that although a single core at a network node can efficiently read or write a single data briquette record, a node cannot read or write so small an amount of data efficiently to another node, at least on present hardware. All messages that are sent from node to node in the code are therefore composed of a concatenation of many data briquette records. These messages are built as grid bricks are updated, and then dispatched at the earliest opportunity as bulk messages. In this way, the messaging can be very efficient. (On Roadrunner, the data transmission speed benefits from frequent prods, via calls to MPI_PROBE, which can cause the messages to move about 30% faster.) In our PPM code, all these messages travel while the communicating nodes continue their computations. If we double the linear size of each grid brick, the sizes of our messages quadruple, while the amount of computation we have to do between message dispatch and receipt is multiplied by 8. Cubical grid bricks 128 cells on a side seem to be sufficiently large to prevent any waiting on message arrival. We note that we do not copy messages upon arrival into any other data structures. Instead, we read the briquette records they contain directly from them as the grid brick update proceeds. Similarly, we write such records into messages for dispatch as a part of the grid brick update, so that the message data is never unnecessarily copied, which would waste limited main memory bandwidth.

Upon each 1-D pass of the algorithm, each CPU core (Cell processor SPU on Roadrunner) updates a strip of grid briquettes in sequence along the entire
length of a grid brick. (This process is described in detail in a later section.) Together, all the cores of a single CPU (Cell processor on Roadrunner) update a single grid brick, with a local barrier between only these cores at the end of this update. The update cannot begin until two messages are received, one for each end of the grid brick in the direction of this 1-D pass. As part of the update, the cores cooperatively build two messages that are sent at the earliest opportunity in opposite directions. On Roadrunner, the messages must first be sent to the host core, and then be sent to the destination node, where they will, upon receipt, be dispatched to the intended Cell processor. On other machines, this message passing is performed by the master OpenMP threads and is simpler, but the result is the same.

The hierarchical structure of the data in our PPM code corresponds precisely with a hierarchical decomposition of the physical problem domain into ever smaller rectangular solid regions. These regions are, from small to large: (1) the grid briquette, (2) the grid brick, (3) the subdomain updated by a team of network nodes, and (4) the entire problem. A single Fortran array giving a global description of the entire problem’s data would therefore have 13 dimensions. Happily, only a few of these dimensions need be exposed in any single subroutine of the code, and we can use Fortran’s feature of allowing data passed as an argument by reference to be dimensioned within any given subroutine in whatever turns out to be the most convenient fashion. This is a very powerful programming technique, but it cannot serve completely to eliminate confusion. A language with a partitioned global address space (PGAS) would therefore be natural for the expression of our computation, but no such language now seems to recognize more than two hierarchical levels of locality – local and not local. Modern machines, and especially Roadrunner, seem to call for a much richer set of distinctions than just this.

The usefulness of our concept of teams of nodes that cooperatively update single, contiguous, rectangular solid subdomains of the problem region is best seen when we consider the problem of parallel I/O. When our process of successive 1-D passes reaches a point at which output is desired, Roadrunner’s host processors are sent complete copies of the grid bricks being updated by their Cell processor accelerators. Without forcing the execution of the grid brick updates to wait, the hosts build from this data highly compressed representations used for visualization and others used in more quantitative analysis. On other machines, this I/O data generation is done in parallel by all the worker cores as part of a special 1-D pass. On either type of machine, the compressed data is then sent to “team leader” processes that run on separate network nodes, where their asynchronous activities will not interfere with continued computations. On Roadrunner, we typically run with 32 Cell processors (256 cores) per team, but on other machines we have run with as many as 4096 team members and 12,288 cores per team.

Restart dumps are written out while the computation continues. On Roadrunner, there is no real cost in holding a memory image of the local problem state on the host while this I/O happens. It is not unusual for hundreds or even thousands of time steps to take place while the restart dump is being written. On other machines, we must reserve extra memory to hold the local problem state for the restart dump. We find that this is not a practical limitation in the operational mode where we use these large machines as engines to run large problems fast rather than as means of running truly enormous problems in weeks or months. Consequently, in this mode we always have plenty of memory available to hold the compressed data that we write out to disk at frequent intervals. There seems always to be plenty of room also for even the far more
voluminous restart data to be held in memory until we are able to write it slowly out to disk. For the two-fluid problems reported on below, in our largest problems we update bricks of only $256^2 \times 128$ cells. Even with Roadrunner’s 1 GB of memory per CPU core, mirrored on the host node, we have no trouble accommodating a new and old copy of the problem data for the update as well as a third copy for writing a restart file.

**Single Processor Performance**

The rate-limiting consideration for our code on every machine to date is single processor performance. If we could get more of the peak performance of each core, our code would run faster by that factor. Neither the latency and bandwidth to the processor memory nor to the other nodes of the machine turn out to be important. This surprising result is a consequence of special code pipelining transformations that we do to exploit the SIMD engines of the Cell processor SPU cores. These transformations work equally well, and in certain respects even better, on all other processor cores, including on GPUs.

Our original intent with these pipelining code transformations was to reduce to a minimum the on-chip data storage space required for the numerical algorithm to run efficiently in vector mode. We found that these transformations gave a dramatic increase in code performance, not only on Cell, but on all other multicore CPUs we have yet tried. A key aspect of this coding strategy is to build grid briquette records in main memory that can be very efficiently read into a core’s cache memory all at once. These then become quanta of data for the algorithm, as explained earlier, so that we never read or write only a portion of such a record. Each core processes grid briquette records in a sequence corresponding to the direction of the present 1-D pass. Records are fetched one entire processing time ahead, and updated ones are written back asynchronously.

The briquette records are unpacked once they are on chip, and before writing them back, we may choose to transpose their contents in preparation for the next 1-D pass. It is an essential point that the briquette processing sequence and the pipelining of the code permits temporary data stored on chip from the processing of one briquette to be used in the processing of the next one in the sequence. This strategy results in a completely pipelined processing of an entire strip of briquettes, which is considerably more efficient than simply processing each briquette in a stand-alone fashion (standard cache blocking). We find the performance benefits of this approach to deliver for our algorithms a factor of from 3 to 6, depending upon the machine. The procedure has been described in more detail elsewhere [6-8] (see also the next section), and we are building automated code translation tools to lessen the significant programming burdens.

On Roadrunner, the inaccurate 32-bit floating point of the Cell processor forces us to use 64-bit arithmetic for some portions of our code, but on other machines we are able to use 32-bit arithmetic exclusively. This is possible because of careful algorithm design. Single processor core performance of roughly 5.5 Gflop/s, or over 23% of the 32-bit peak, is achieved on Intel Nehalem CPUs with 2 threads per core. With only one thread per core, performance drops to 4.38 Gflop/s/core @ 2.83 GHz, and this performance is observed averaged over the entire code at a scale of 8192 cores. The need on Cell to perform a mixture of 32-bit and 64-bit arithmetic causes the code performance to drop to 1.4 Gflop/s/core, and we have observed 1.37 Gflop/s/core when running the Roadrunner code on 7168 Cell processors (and 55,296 SPU cores).

**Scaling Studies.**

We have developed a detailed performance model for our code. Results for a recent fairly large scaling test run on
Roadrunner are as follows. Each grid brick had $128^3 \times 256$ cells in $64^2 \times 128$ briquettes, and each core updated 4 of these briquettes at a time (Fig. 1) at a sustained rate of 1.37 Gflop/s, averaged over the whole run with all costs of the simulation included. Each 1-D sweep took 0.554 sec., and the messages sent in Y during the first X-pass were the largest, at 28.33 MB sent in each Y-direction as a single message, while those sent in X or Z in other 1-D passes were only 10.88 MB each. The bi-directional interconnect link bandwidth needed for these asynchronous messages to reach their destinations without causing processors to wait for them was only 205 MB/sec, which is very modest for any large machine today. All messaging to I/O processes had enormous amounts of time to arrive, since we produced data dumps only every 512 time steps, or every 14.2 min. The largest messages sent to I/O processes were for restart dumps, which we produced every two dump intervals, or about every 28 min., and we observed that these dumps took only 7.5 min. to write, with 256 I/O processes writing to disk in parallel. Each of these team leaders had a team updating $3^3$ grid bricks, and we had 8 such teams in X and in Z but only 4 such teams in Y, for an overall grid of $3072^2$ cells. The requirement for bandwidth to each core from its main memory is determined as follows. 4342 flops (with reciprocals counted as only 3 flops) are performed per grid cell per time step. We read in 16 briquettes and write back 4 on each 1-D update of 4 briquettes involving $4^2 \times 2 \times 1447$ flops. Since each briquette record is 480 bytes, this amounts to 4.82 flops/byte/core. Our observed computation rate of 1.37 Gflop/s/core then implies a utilized main memory bandwidth of 0.284 GB/sec/core. Thus we see that our code places only very modest demands upon the Roadrunner interconnect and memory system. We conclude that its performance is determined entirely by single core computational performance. This explains its near perfect scaling to 7168 Cell processors.

Lessons Learned from the Roadrunner Code Implementation.

In this work we decided at the outset to implement the entire computation, except for MPI messaging and I/O data preparation, on the Cell processor SPU cores. This caused some problems during the course of the work, because we were then forced to reduce the data workspace in the on-chip store to an extreme level. Nevertheless, we feel that this decision was sound, because our multifluid gas dynamics computation does not break down easily (if at all) into a sequence of small "kernel" computations. We can indeed identify small computations, such as the determination of an interpolation parabola to describe the behavior within a grid cell of a particular fluid state variable like the pressure. These small computations are logically distinct, and they can easily be encapsulated in separate subroutines. However, they are tied into the other computations we perform in such a way that we are forced to provide multiple inputs and outputs for each such kernel. This fact diminishes the computational intensity of these kernels, because they are forced to read and write quite a lot of data relative to the amount of work they perform.

This matter of computational intensity can be quantified for our multifluid PPM codes. Doing this reveals a memory cliff, which occurs at quite a specific range of computational granularity. Our Cell processor implementation is on the right side of this memory cliff, but it is arguable that present day GPUs, like Nvidia’s Fermi, are on the wrong side of it.

<table>
<thead>
<tr>
<th>&quot;Kernel&quot;</th>
<th>flops/word</th>
<th>KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 time steps</td>
<td>250</td>
<td>16,000</td>
</tr>
<tr>
<td>1 time step</td>
<td>130</td>
<td>12,000</td>
</tr>
<tr>
<td>1-D pass</td>
<td>43.3</td>
<td>256</td>
</tr>
<tr>
<td>1 parabola</td>
<td>4.25</td>
<td>64</td>
</tr>
<tr>
<td>Riemann states</td>
<td>2.26</td>
<td>64</td>
</tr>
<tr>
<td>fluxes</td>
<td>3.12</td>
<td>64</td>
</tr>
</tbody>
</table>
This table takes some explanation, but at first glance it is clear that when we allow ourselves about 256 KB of on-chip memory (per core), the computational intensity that we can achieve goes up by an order of magnitude. This is a very important fact that has significant implications for the design of multicore processing chips.

In the table above, "kernel" refers to an easily separated and bundled portion of our multifluid PPM algorithm. In our Roadrunner implementation, each 1-D pass of the algorithm for an entire grid pencil of $4^2 \times N$ grid cells, with $N$ typically 128 or 256, is performed by a single CPU core in a single subroutine containing a single vector loop (see the next section). This entire computation occurs with all temporary results stored on chip and with no extraneous data traffic between the CPU core and its main memory. As the table entry indicates, there are 43.3 flops performed for each word of data that is either read from or written to the off-chip memory. If we do 3 such 1-D passes on a grid brick of $32^2$ cells stored on the chip, then we need 3.66 MB each for an old, a new, and a prefetched brick, plus about 120 KB for the program instructions. All this would need to reside in a shared cache on the multicore CPU, and each core would need its own private cache of 256 KB to perform its grid pencil updates. In this implementation, we would need a 12 MB L3 cache with multiple 256 KB L2 caches, one for each core. Such devices are presently available. This strategy would allow us to do 130 flops for each off-chip word read or written.

Going down in the table, we see that when we try to decompose the 1-D pass grid cell update procedure into a sequence of separate kernels, we achieve a modularity to the numerical algorithm but we find a collapse of the computational intensity. The table entry "1 parabola" is a good example of this very important phenomenon. This table entry refers to a single subroutine in the PPM code that interpolates a parabola to represent the variation of a single state variable, such as the $y$-component of the velocity, within a grid cell. Values of the variable in neighboring cells are utilized, and 3 coefficients are produced describing the parabola that are used elsewhere in the gas dynamics computation. For the most elaborate interpolation that we use in the code, which takes the most computational labor, 8 numbers per cell must be read or written and only 34 flops are performed. The computational intensity is therefore quite low. If, however, we implement this algorithm in an advection scheme that moves this single variable through our Cartesian grid according to a prescribed velocity field that we can compute from a formula, then most of the input and output values to this interpolation algorithm become intermediate results that need not be stored in off-chip memory. The computational intensity is then 4 times greater, and we can make this computation proceed on standard multicore CPUs today at roughly 25% of their peak 32-bit performance. This performance is nevertheless misleading, because we cannot implement this single portion of our gas dynamics algorithm usefully in such a stand-alone fashion. As a part of the larger computation, its computational intensity is 4 times lower. This does not matter if the entire computation is performed from data resident on the chip, but it does indeed matter if the chip does not have sufficient memory capacity to support this. This appears to be the situation with today's GPUs.

Our decision to implement our entire gas dynamics computation for execution on the Cell processor SPU core had benefits in addition to freeing us from constraints of limited main memory bandwidth to the CPU. It also eliminated concerns of detailed synchronization between the Cell processor and its host CPU as well as eliminating the need for potentially massive data transfers between the memories of these two devices. We found that the host CPU was free to asynchronously do
all the MPI messaging, output data generation, and restart dump I/O. In this way we were able to eliminate many of the serial sections of code that might otherwise have slowed our computation due to the implications of Amdahl’s Law. The cost for this approach was one host core for every 8 worker cores and of course the otherwise unused host memory. On machines other than Roadrunner, we assign the I/O labor to only a very small number of extra cores and we have the MPI messaging performed by MPI’s system thread, which seems to cost, amazingly, about 6% to 12% of the running time even though all the messaging is fully overlapped with computation.

The remaining important lessons we learned from our implementation of multifluid PPM on the Roadrunner machine concern the benefits of vectorized computation in a pipelined fashion with very short, aligned operands. This is discussed in the next section.

**Details of the Update of a Single Grid Pencil by a Single CPU Core**

Our implementation of the multifluid PPM algorithm on the Cell processor was the result of two constraints: (1) the Cell SPU offers only a 4-way SIMD engine for arithmetical computation and (2) the Cell SPU has only 256 KB of on-chip memory. These two features left very little room for alternative strategies. Because standard CPUs from other vendors also offer, although not exclusively like Cell, a 4-way SIMD engine and only 256 KB of L2 cache per core, it turned out that coding to the two above constraints produced very good code for all CPUs. This was an unexpected benefit. Other CPUs allow one to write less constrained code, but for our applications this seems to produce no significant performance benefit. Thus we find that a single code expression can, with appropriate compilation, produce excellent code for all CPUs manufactured today, at least in our fluid dynamics domain. We cannot rigorously prove this assertion, but we strongly suspect it to be true.

Coding to the above 2 constraints, we came up with the following implementation. Each Cell CPU updates a single grid brick composed of grid briquettes. This operation consists of a number of grid pencil updates, which are performed in parallel by the cores of the CPU. A grid pencil is a strip of grid briquettes extending all the way through the grid brick in the direction of the 1-D pass, as shown in Fig. 1. In the figure, the grid pencil is composed of 4 strips of grid briquettes, with a 2x2 set of grid briquettes at each longitudinal position. Because our most general PPM algorithm requires transverse data even in a 1-D pass, at each longitudinal position in our traversal of the grid pencil we read into the on-chip cache 12 grid briquettes – a 2x2 set of briquettes to update plus 8 more briquettes holding data offset in the transverse dimensions. This transverse data is not shown in Fig. 1, which corresponds to a simpler version of our code that is restricted to problems in which no shocks of more than Mach 2 occur. The diagram appropriate to our more general code versions is shown in Figure 2. The logical flow of the grid pencil update code executed by each CPU core is as follows:

1000 continue
   prefetch the next 12 briquettes.
   unpack the previous 12 briquettes.
   construct aligned 16-word vectors.
   do 2x16x1447 vector SIMD flops.
   pack results into 4 new briquettes.
   write the 4 new briquettes back.
   if more briquettes in strip, go to 1000.
Return to begin next grid pencil.

From this control flow it should be clear that all accesses to off-chip data are strict sequential transfers of entire grid briquette records (of 480 bytes each). For the Cell SPU, the prefetch is explicit, while for other CPU cores it is implicit. We express a prefetch to the cache in standard Fortran as follows. All variables and arrays that
Figure 2. The grid pencil update process by a single CPU core for the more general PPM codes. At the lower right are 4 “real” (red) and 8 “ghost” (blue) briquettes (of $2^3$ cells each) that are extracted into the on-chip cache. At the lower left are 4 updated briquettes to be written back into the new grid brick data structure.

we intend to reside in the on-chip cache are declared locally within a subroutine, so that they are on the subroutine stack. The stack will be cache resident if it fits in this small memory, and, just as for the Cell SPU, we are careful to add up its size and guarantee that it will indeed fit. A data prefetch is also easily expressed in Fortran. We simply do a data assignment loop, copying a portion of a main memory resident array (specified as a subroutine argument) into a locally dimensioned array on the stack. This will be a fetch if we immediately use the copied data, and it will be a prefetch if we do not use this data until, for example, the subsequent iteration of a large loop in which it occurs. This data copy may seem unnecessary, but it is not. If we use this data in main memory in any way in our routine, the hardware will force a copy into the cache. We have simply expressed this copy explicitly. This expression has significant benefits when multiple cores work out of a commonly shared main memory. The hardware will assign locations in main memory to our cache resident data. These main memory assignments will be private. Consequently, when we work with the shared data there will be no needless cache coherency traffic generated, and we can proceed at full speed. Copying the data into the cache cannot be avoided, whatever code expression we use. Needless copying the private versions of this data back to main memory is avoided by continually reusing the on-chip data arrays so that the write-back caching policy will essentially never write any of this data back to main memory. The reason that write backs essentially never occur is that we have taken care to see that all our data arrays on our stack fit into the cache. Thus there is never a need to spill any of this data to main memory to make room for new values. We simply write over the same temporary values again and again.

From the above description one essential feature of this grid pencil update does not yet clearly emerge. The computation is thoroughly pipelined. This turns out to be necessary to make the computation efficient at the tiny granularity of a single grid briquette. Pipelining is a familiar concept from vector processing. In that simple case, an arithmetical operation is decomposed into a handful of stages and all stages execute simultaneously on different operands working their way through the pipeline. Here, however, we talk about pipelining at an extremely high level. Instead of pipelining a single add or multiply, we pipeline the entire grid cell update process, expressed as thousands of lines of Fortran code. The pipeline is expressed as a humongous Fortran loop, with its loop index specifying the index of the plane of grid briquettes that we operate upon in going down the grid pencil as shown in Figure 2. In our code, an automated code translator in-lines all the many subroutines that express the whole gamut of operations that we must
perform in sequence in order to arrive at a fully updated grid briquette. Each operation is performed, in our latest code expressions, on all grid cells of the 4 grid briquettes shown in red in Fig. 2. This is achieved via vectorizable loops that range over these cells. Our code translators automatically generate SPU vector SIMD intrinsic function calls for the Cell processor, SSE intrinsic function calls for Intel or AMD processors, or Altivec intrinsic function calls for IBM PowerPC processors. These SIMD operations specify groups of operations that occur simultaneously, so that a vector of length 16 or 32 involves 4 or 8 of these calls. Thus these loops vanish in the translated code expression, and only the single, humongous loop over planes of briquettes survives. In one iteration of this big loop we perform every operation necessary to fully update a grid cell, but we do not perform each such operation on the same cell. Instead, we have a whole set of planes of grid cells in process in this pipeline at once. In general, the number of such grid planes in process is related to the difference stencil of the numerical algorithm. For our present multifluid PPM code, we have 7 such grid planes in process at once. If we use vectors of length 64, so that we process 4\(^3\) cells simultaneously in the pipe, then we will have 10 grid planes in process.

It should now be clear how we make this whole computation fit into our tiny on-chip cache. We only need room on chip for the grid planes of data that are active in our pipeline. As we pointed out above, for PPM this is at most 7 or 10 grid planes of 64 bytes each per variable. Many of the intermediate results generated at one pipeline stage are consumed shortly afterward, one or two stages further down the pipe. These require much less on-chip memory. In fact, the total requirement for our two-fluid PPM code is about 60 KB. This is a truly miniscule amount of working data. The computation is efficient only because it is pipelined. We are constantly bringing new data into the beginning of the pipeline and sending it off to main memory from the end. The computation never has to stop to go fetch fresh data or to write back updated results. Because every single instruction is a 4-way (or in general m-way) SIMD operation on aligned operands, and because so many of these instructions are exposed at any stage of the pipeline, this computation can proceed at the greatest rate of which the hardware is capable, essentially never stalling. In principle.

Either the hardware is capable of less than its advertised performance rating or there is some flaw in the above argument. We achieve only about 23\% of the rated 32-bit peak floating point performance, while the above reasoning indicates that we should achieve about 50\% (the fused multiply-add operation is rarely useful in our computations). This is a mystery. We are pleased with the performance we achieve, but would of course prefer double that value. We have recently begun to work together with Jim Greensky at Intel to investigate this matter. If we can identify the cause of this slow-down, then we can perhaps fix it in our code expression or Intel can perhaps fix it in the hardware.

Implications for ASC Project Codes.

One motivation for the work reported here was to test the conjecture that a highly efficient gas dynamics ("single physics") code could be built with a modest level of effort (1 faculty member and 2 graduate students) to fully exploit the unique Roadrunner machine at scale. We consider that demonstration successful, with the single exception that we were forced, despite considerable algorithm redesign, to use 64-bit arithmetic for the multifluid fractional volume tracking because of the fifth-order formal accuracy of the PBP advection scheme. This proved unnecessary on other CPUs, and is hence attributable to the lack of proper rounding for the 32-bit arithmetic on the Cell processor. The fully 32-bit version of our code that
runs beautifully on other equipment. Runs on Roadrunner at 3.9 Gflop/s/SPU-core. Unfortunately, on Roadrunner it does not get quite the right answer, because symmetries that are well preserved for long durations when the code is run on Intel processors become violated relatively early on when this code is run on Cell processors. Presumably the systematic errors in 32-bit rounding cause this effect. When the same code is compiled for both processor types using 64-bit arithmetic, the results are essentially identical.

At the outset of this project, a means of translating performance gains from this work to ASC project codes like XRAGE was set out. Those codes perform a cell-by-cell AMR (adaptive mesh refinement) by extensive use of indirect addressing. This technique does not map well onto the 4-way SIMD operations of the Cell processor SPU. However, we pointed out that if each grid cell of an XRAGE calculation were refined to produce, say, a grid briquette of 4\(^3\) or 6\(^3\) cells, then each of these could be updated in the same pipelined fashion that we use in our Roadrunner PPM codes. Because each grid briquette record can be fetched from any location in main memory with the same speed, it would not matter if the list of such briquettes is essentially unstructured. When grid refinement levels change from one XRAGE cell to another, this would correspond to refinement levels changing from one uniform grid briquette to another. The necessary computations at the interface could of course be performed in SIMD vector mode, exclusively using aligned 4-word SIMD operands. Such an implementation of our PPM multifluid algorithms would take our present codes to the next level of usefulness and complexity, while keeping their single-physics character intact. Such an effort has been very recently begun.

The very simplified messaging of our present Roadrunner code, with only 2 messages received and dispatched per grid brick in each 1-D pass, is not appropriate to an AMR code. The AMR implementation we envision would still use grid bricks that are strict rectangular solids, but these would be smaller than those we now use, and they would be adaptively refined (i.e. not at a single grid refinement level). We have developed a new multifluid PPM code, not yet ported to Roadrunner, that employs a messaging strategy that is more efficient but significantly more complex. In this code, each brick sends and receives 6 messages upon each 1-D pass, and there is a special, rather complicated ordering of the message receives and sends. This strategy would be appropriate for an AMR code, and should permit us to maintain our present overlapping of communication and computation in such a more demanding context. The small average size of the grid bricks one would choose to employ in such an AMR code would permit speculative prefetching of additional grid bricks by each node on each 1-D pass, and this would in turn allow a natural mechanism for automatic dynamic load balancing.

**PPB Fractional Volume Advection.**

Our multifluid PPM codes employ the PPB moment-conserving advection scheme to track the fractional volumes of the fluids within each grid cell. This method was implemented in 2003-2004 by Woodward and released in a module he constructed for the XRAGE code in 2005. PPB is a multi-D generalization of Bram van Leer's 1-D Scheme VI from the 1970s [9]. That scheme was enhanced for constrained advection and generalized to 2-D by Woodward [3], where the enhanced scheme was named PPB (for Piecewise-Parabolic Boltzmann scheme). In the present version, the scheme has been streamlined for efficient directionally split operation, and its constraints have been greatly improved from the version in [3]. Here we describe very briefly the principal ideas of this scheme and give examples of its behavior on the Rayleigh-Taylor type problems that lie at the core of this study.
Figure 3. The action of the PPB fractional volume advection scheme in 1-D (see text).

The PPB advection scheme is ideally suited to treating gas-gas multifluid interfaces in a highly efficient SIMD vector mode on modern CPUs. This efficiency comes from the following properties of the scheme: (1) the scheme is directionally split into 1-D passes, (2) the fractional volume in a grid cell is considered a continuous variable that is non-vanishing everywhere in the cell if it is non-vanishing anywhere there, (3) the scheme achieves high resolving power from its use of 10 independent subcell moments of the distribution in each cell.

The first property on this list makes the PPB scheme highly computationally efficient, not only by simplifying the scheme considerably while preserving its fifth-order formal accuracy, but also by strictly limiting the difference stencil and thus the main memory bandwidth requirement. In our version of PPB, we give up strict fifth-order formal accuracy, since formal accuracy in our problems and actual accuracy are only rather weakly related. This makes the scheme considerably simpler and more computationally efficient.

The second property on our list is very important. In each 1-D pass, we represent the distribution of the fractional volume by a parabola, as shown in Figure 3. The parabola is determined from 3 moments of the distribution in the cell. Although PPM also uses parabolae for interpolation, it has much less resolving power in practice. This can be understood by the simple fact that in 1-D PPM works with 3 times less information than PPB, and it is therefore not surprising that PPM, as a general rule of thumb, needs about 3 times as many grid cells in each dimension in order to match the quality of PPB advection. This difference in resolving power is not immediately apparent in a simulation, but becomes clear as the computation is carried forward for an extended simulated time. Van Leer’s original paper [9] explained this behavior (see also [3]) by pointing out that the fifth-order character of PPB comes from its small rate of error accumulation rather than from its ability to very closely fit any particular distribution on a grid.

PPB’s small error accumulation is a very interesting feature of the scheme. PPB does not represent a sharp multifluid interface as an infinitely thin line, as do various methods in frequent use today. In fact, there are many problems that arise from trying to represent any infinitely thin feature on a grid of cells with finite width. (See for example the discussion in [10] of the “Cray instabilities” for strong shocks.) With the PPB scheme we take the view that we can afford to smear the multifluid interface out over one or two grid cell widths as long as this interface thickness does not grow significantly, if at all, during the course of a simulation. This approach immensely simplifies the treatment of an interface, because it eliminates the large number of separate cases that we would otherwise need to treat each in its own appropriate way. There is even a “straightforward” generalization of the method to handle any number of fluids in a single cell,
based upon the simple fact that if we add two parabolae we get another parabola. The only complexity arises from the constraints that we must apply to these parabolae to satisfy the common sense requirements that the fractional volume of a fluid in a cell cannot exceed unity, nor can it assume negative values. We use two ways of applying such constraints, as illustrated in Figure 4. At the top in the figure is the method presented in [3]. This method is not appropriate when the parabola to be constrained is located next to a cell in which the fractional volume is either unity or zero. In such a case, the method illustrated in the lower panel of the figure, similar to the method used in PPM, is more appropriate.

Constraints to the parabolae used in the PPB scheme are essential in our problems. They must be very sparingly and carefully applied, because otherwise the accuracy of the scheme can be destroyed or various unwanted results produced, such as tiny pockets of fluid A just inside a sharp interface with fluid B. From the illustration in Figure 3 of the operation of the
PPB scheme, it is clear that the method should be useful in describing gas-gas mixing, in which no surface tension or material strength effects exist. However, the nonlinearity and complexity of flows near unstable multifluid interfaces makes it difficult to predict how any method will work in practice. Therefore we illustrate the action of the PPB multifluid interface treatment in our PPM codes through examples.

A simple example showing the resolving power of PPB advection is given in Fig. 5. In each grid cell, the internal distribution determined by PPB’s 6 conserved moments is shown. In the bottom panel of Fig. 5, a 16x16 grid was used to compute the advection of a swirling flow beginning with the distribution shown in the top panel. The distribution in the middle panel shows the result of the scheme on a fine grid. The PPB representation on the 16x16 grid is remarkably good, given the coarseness of the grid and the preservation of features no more than one cell wide. This kind of swirling flow is frequently encountered in turbulent regions, and hence this simple 2-D test problem is quite a meaningful way to see the benefits of PPB advection.

**Single-Mode Rayleigh-Taylor Instability Test Problem.**

In our work, we have made extensive use of Guy Dimonte’s single-mode Rayleigh-Taylor test problem to refine our numerical methods, discover the good and unfortunate features, and to devise mitigation strategies for the latter.

First, it is important to realize that, like shocks, multifluid interfaces have a physical mechanism that causes them, when they are unstable, to steepen. Because these interfaces are so often unstable during the course of any given problem evolution, we can think of them as physically steepening most of the time. The cause of this physical steepening is illustrated in the single-mode Rayleigh-Taylor (RT) problem. As the instability proceeds, the surface area of the multifluid interface increases without bound. Because the multifluid mixing fraction, our fractional volume variable, is a constant of the motion, the volume of the mixed region should remain constant. As a result, the thickness of the interface must decrease.

The steepening of multifluid interfaces poses numerical difficulties, just as does the steepening of shocks. A major difference, however, is that shocks are, quite generally, strongly stable; multifluid interfaces certainly are not. With our PPB representation of the interface as a continuous transition from 0 to 1 in the fractional volume variable, $f$, we naturally initialize the problem with this transition resolved on our grid. However, after very little time, the interface becomes as thin as it can be in our numerical representation. For PPB, this smallest thickness is extremely thin. Plotting it, one has the impression that it is only about one cell wide, although of course this is impossible. Actually, our smooth representation with PPB of the interface transition is a boon to the numerical treatment, because it tends to eliminate all manner of numerical glitches that might otherwise occur. Nevertheless, there is a necessary trade-off between the thinness of the numerical representation and the excitation of interface perturbations through the necessarily slightly changing representation of the transition as its center moves from a grid line to the center of a cell. Such perturbations are invisible in the results unless the interface is unstable, which it so often is. They arise at beat frequencies between the interface representation and the grid, with the fundamental excited wavelength usually at the frequency at which the thin interface crosses grid lines. This effect is very similar to the mechanism for shocks that gives rise to the “Cray instabilities” identified in [10]. The solution is the same, namely to spread the interface representation out so that the amplitude of the perturbations is diminished. These perturbations are nevertheless unavoid-
Figure 6. Single-mode RT test with initial interface thicknesses $\delta$ of $0.1\lambda$ (top) and $0.05\lambda$ (bottom).

Figure 7. Single-mode RT test with initial interface thickness $\delta = 0.05\lambda$ and $\mu = 10^{-5}$.

setting the initial density and sound speed in the light fluid at the interface to 1 also, the units of length, time, and mass in our problem are defined. In these units we set the gravitational acceleration to 0.024, following the RT study by the “alpha group” [11]. This choice results in a fairly incompressible flow. We use a gamma-law equation of state with $\gamma = 5/3$ for both fluids. In the visualizations of the flow shown in Figs. 6-8 and in following figures, we use a volume rendering technique to display the volume fraction, $f$, of the denser fluid. Pure dense fluid is a rather transparent red, mixtures in which this fluid predominates vary from red to yellow, white represents 50-50 mixtures, aqua and finally dark blue represent mixtures with greater amounts of the light fluid. The pure light fluid is made fully transparent.

In Fig. 6, the computational grids are $256^2 \times 3072$, and our code solves directly for the perturbation to the unstable equilibrium state with a flat interface. This procedure allows us to get the most out of 32-bit arithmetic. The flow is shown at time 30 in two calculations that differ only in the extent of smearing of the initial interface. The initial transition from 0 to 1 for the fractional volume $f$ of the denser fluid
is prescribed as \( f = .5 \left[ 1 + \sin(ny/\delta) \right] \)
where \(|y| \leq \delta/2\) is the distance from the middle of the transition in the vertical direction, measured in perturbation wavelengths (\(\lambda\)) and \(\delta\) is a measure of the thickness of the transition region.

Because of the properties of the sine function, as a practical matter the interface thickness is more like \(2\delta/3\). For the two runs compared in Fig. 6, this initial interface thickness is 17 (top panel) and 8.5 (bottom panel) grid cell widths. These values may at first seem excessive, but we must realize that the displacement amplitude that we are required to resolve on our grid in this problem is just \(5.12\) grid cell widths. The peak-to-trough displacement is \(10.2\) cell widths, just comparable to the interface thickness in the case shown in the top panel of Fig. 6.

The initial smearing of the multifluid interface transition is a regularization of the problem that allows the subtle displacement of the interface to be well resolved and well treated on this grid. It also allows the amplitude of the disturbance to grow substantially before the physical thinning of the interface as its area increases causes it to reach the thinnest possible numerical representation that our PPB scheme permits. This avoids the introduction of tiny beat frequency perturbations at low disturbance amplitude, when these beat frequencies would be many grid cells in wavelength and hence lovingly followed by our very accurate PPB scheme. The multifluid interface in the simulation in the top panel is very smooth, due to these regularizing effects, while that in the lower panel shows secondary unstable modes of both the Rayleigh-Taylor and Kelvin-Helmholtz varieties. These secondary instabilities have been introduced at wavelengths of many grid cell widths, because shorter wavelengths are, gratefully, damped by the numerical scheme’s dissipation.

Although the flow in the top panel of Fig. 6 is arguably more beautiful than that in the lower panel, this does not mean that it is more correct. Both simulations solve the inviscid Euler equations, and consequently the secondary instabilities should definitely grow, if there are any corresponding perturbations at all. It is hard to imagine how one could build a laboratory experiment corresponding to these simulations, with negligible viscosity, and introduce only the single perturbation that we prescribe for the initial state. It is thus not clear how the growth of secondary instabilities could possibly be avoided in the real world. One could say that an inviscid simulation that did not show such features at some level would simply not be credible as a representation of real fluid behavior.

In both these simulations in Fig. 6, as well as in the others we present in this article, we have introduced an additional technique intended to reduce the specialness of particular locations in the flow, such as grid plane crossings of thin multifluid interfaces or planes of symmetry. This technique is grid jiggling, which we used in [10] to deal with Cray instabilities of slowly moving strong shocks. The idea is that we can solve the problem equally well in any inertial frame of reference. By solving it alternately in frames that move in one direction and then in the opposite direction in the horizontal plane, we eliminate much of the specialness of otherwise special locations. Since the fluid is moving anyway, our simulation is equally accurate in any moving frame, as long as we do not move the frame of reference faster than the average fluid motion. In our simulations presented here, we solve the fluid equations in frames moving at horizontal velocities of ±0.025. Remembering that the sound speed is unity, we see that we never move our reference frame in any direction more than a \(40^\text{th}\) of a grid cell width. This grid jiggling has essentially no effect upon the formal accuracy of the calculation. At the location of a very sharp multifluid interface, however, it will have a non-negligible effect, because the representation of
the transition in \( f \) across the interface cannot be truly smooth there. This technique thus introduces a tiny dissipation that has a very high formal order but that can be important at interfaces that have become thinned to the maximum allowable extent.

The flow shown in Figure 7 gives the result of adding one further physical regularization to this problem. In this simulation, we introduce a viscosity but do not include any associated heat conduction. In our units, the dynamic viscosity, \( \mu \), is \( 10^{-5} \), which is indeed very small. Nevertheless, the effect upon the secondary instabilities at the time shown is signi-
significant. Comparison with such simulations on coarser grids suggests that this flow in Fig. 7 is very nearly converged, which we might hope to be possible for the Navier-Stokes equations we are solving here. Nevertheless, at later times the secondary instabilities do indeed occur in this simulation, because the interfaces are all unstable and the viscosity is very small. A comparison of these same 3 simulations at times 40 and 50 is shown in Fig. 8, covering the time interval in which full-scale turbulence develops in all 3 simulations. Our regularizations of the problem can delay the onset of turbulence, but they cannot prevent it unless we very significantly increase the value of the viscosity. It is interesting to note that in the 3 simulations compared in Figs. 6-8, the different regularization strategies employed have rather little impact upon the computed penetration of the dense fluid into the light fluid, although the strength of the secondary instabilities varies considerably between them. If it is this penetration that we are interested in, we may perhaps regard the various regularizations just discussed as having effects that are more cosmetic than physically significant.

**Results for Multimode Rayleigh-Taylor instability.**

To investigate the role of initial conditions on the spreading of an RT-generated mixing layer, we have so far studied two cases with fluids of densities 3 and 1 (Atwood number $\frac{1}{2}$). We again set the density and sound speed of the light fluid at the mid-plane to unity, use $\gamma = 5/3$, set the width of the problem domain to unity, and use a gravitational acceleration, as in the alpha group study [11], of 0.024. The narrow-band initial disturbance spectrum is motivated by the alpha group study but exploits the fine grids we can use with our code to allow more nonlinear disturbance wavelength doublings. In the initial state, smearing the multifluid interface with a thickness $\delta = 0.0015625$, we confine the wavelengths of disturbances to the range between 1/128 and 1/256 of the domain width, and we set all permitted disturbance mode amplitudes to random values, where the average such value has no dependence upon mode frequency. The rms amplitude of the interface displacement is 4% of the longest wavelength allowed, namely $3.125 \times 10^{-4}$. The idea, as in previous such simulations, is for the precise nature of our initial disturbance to become quickly irrelevant as the many high frequency modes interact nonlinearly to produce ever longer wavelength disturbances. Ultimately, since we introduce no dimensional parameter such as a viscosity, other than the value of gravitational acceleration, the flow development must become self-similar. This expectation depends upon the size of our periodic domain, the size of the scale height in either fluid, and the size of the speed of sound being so large that they act, as a practical matter, as if they were infinite.

We wish to study the rate at which the developing mixing layer spreads in the vertical dimension and to measure the coefficient, $\alpha$, that describes this quantitatively. To accomplish this, we create horizontal averages of the mixing fraction, our fractional volume variable, $f$, as the code runs. We define the height, $h$, of the mixing layer as the contour where $f = 0.99$. Because the gravitational acceleration, $g = 0.024$, is the only dimensional parameter in the problem (all others, as noted above, are regarded as either very large or very small), the mixing layer thickness, and hence $h$, must scale with time, $t$, as $gt^2$. Introducing the scaling with the dimensionless Atwood number, $A$, from linear theory, we may write in the self-similar regime that $h \sim \alpha_s Agt^2$.

To observe this self-similar scaling unequivocally in a numerical simulation, it is useful to make the domain large enough to contain a huge number of rising bubbles, so that the results will not be influenced appreciably by any single one, even after several doublings of the mixing layer thickness. Using the Intel-
processor-based HP machine of the Minnesota Supercomputer Institute, we employed a grid of \(4096^2 \times 1024\) initially, later rezoning this to produce a grid of \(2048^2 \times 4096\) cells by coarsening in the horizontal dimensions and adding cells in the vertical. This fine grid allowed us to confine all modes making up the initial disturbance of the unstable interface to wavelengths between \(1/128\) and \(1/256\). We were still able to resolve the finest initial modes with 16 grid cells. The result was that we had so many rising bubbles that statistical effects in our determination of the bubble front height, \(h\), were minimal over a major portion of this run. The degree to which we have resolved the initial development of the instability can be seen in the zoomed in images of a thin slice taken diagonally through the mixing layer at times 1, 1.5, and 5.5 in Figure 9. The coarseness of our representation is evident at the two early times, but by time 5.5 our grid is certainly adequate for resolving the much larger wavelength modes that have developed.

In the second portion of Figure 9, we show images of the central diagonal slice through the problem domain in a manner that permits a visual check on the self-similarity of this flow. According to the scaling law, the mixing layer should double in thickness for each factor of 1.4 in time. Thus if we show the central diagonal slice at times separated by this factor zoomed out a factor of 2 each time and with double the thickness each time, all these images should look essentially alike. We show just such images at times 11, 17, and 25.5 in Fig. 9. These time intervals do not quite correspond to the scaling relation, because we chose those output times for which the layer thickness had nearly doubled, and the self-similarity of the flow is not perfect. Nevertheless, these 3 images do indeed look very much alike, as self-similarity suggests that they must. The extent that textbook quality self-similarity is achieved in this simulation can be judged from the log-log plot in Fig. 10. By the end of the run, we are pretty much there, although by this time the flow is dominated by a relatively small number of huge plumes, and statistical variations begin to complicate the self-similar trend. The results of this simulation give a value for the scaling parameter \(\alpha_{\text{bubble}}\) of 0.030, consistent with the alpha-group study in [11] and with the high-resolution simulation of that problem in [12], but with less statistical variation in the data.

The simulation shown in Fig. 9 was run at the Minnesota Supercomputer Institute due to special availability of their machine during its initial testing phase. The bulk of our work has been performed on Roadrunner, which is capable of much larger runs. As an example of several such runs that we have carried out to date, we present results of a Rayleigh-Taylor mixing simulation that differs from the run just described in that it has a broadband initial spectrum of disturbances of wavelengths ranging from \(1/3\) to \(1/1024\). Again, the amplitudes of the modes in this range are chosen randomly, but now the average amplitude of a mode of a given wavelength, \(\lambda\), is proportional to \(\lambda^{-2}\). The rms amplitude of the overall initial disturbance was set to \(3.125 \times 10^{-3}\), ten times larger than for the run in Fig. 9. The initial grid was \(8192^2 \times 1024\), and the problem was run on 8448 Cell processors. As the mixing layer spread, we coarsened the horizontal grid while adding more cells in the vertical, to produce a grid of \(4096^2 \times 2048\), and finally a grid of \(2048^2 \times 4096\) cells. The time dependence of the bubble front height, \(h\), is shown in Fig. 11 with results of the alpha group and of our code running on the alpha group grid and initial data included for comparison. Clearly, this initial data gives rise to a much more rapid spreading of the mixing layer, similar to results seen in laboratory experiments and consistent with the model in [1]. Volume rendered views of the developing mixing layer are shown in Fig. 12.
Figure 9. Narrow-band multimode RT simulation at times 1.0 (top), 1.5 (middle), and 5.5 (bottom). We view a central slice of the problem domain 0.025 thick. At times 1 and 1.5, we see the central 16\textsuperscript{th} of this slice, while at time 5.5, we see the central 8\textsuperscript{th} of a diagonal slice. On the next page, we show this same simulation at times 11, 17, and 25.5, viewing the central 4\textsuperscript{th}, half, and all of the diagonal slice of thickness .025, .05, and .1, respectively.
Figure 10 (left). Log-log plot of bubble height, \( h \), as a function of \( \text{Ag}^2 \) for the simulation shown in Fig. 9. The two straight lines of slope unity show trends for \( \alpha_{\text{bubble}} = 0.025 \) and \( 0.035 \). The simulation gives \( \alpha_{\text{bubble}} = 0.030 \).

Figure 11 (right). Plot of the bubble height, \( h \), as a function of \( \text{Ag}^2 \) (L, the size of the domain, is unity) for the broadband simulation discussed in the text and shown in Fig. 12 (black, upper curve), for our code (red, middle curve) and for Youngs' code (blue, lower curve) running the alpha-group problem as specified in [11]. The broadband run gives \( \alpha_{\text{bubble}} \approx 0.06 \).

Figure 12. Broad-band multimode RT simulation viewed from the side shortly after the first grid coarsening from \( 8192^2 \times 1024 \) to \( 4096^2 \times 2048 \) cells. On the next page, the subsequent development of this mixing layer is shown. These images are taken from a movie available on the Web at [www.icse.umn.edu/MOVIES](http://www.icse.umn.edu/MOVIES). See the text for a discussion of this run.
Conclusions
We have developed a multifluid PPM gas dynamics code intended specifically to run and scale well on the Roadrunner machine at Los Alamos. The code includes a powerful new method, PPB multifluid volume fraction advection, for the numerical treatment of problems with unstable gas-gas interfaces. This code and the scale and power of Roadrunner have allowed us to carry out simulations of Rayleigh-Taylor mixing on very fine grids that permit us to resolve an enormous range of scales in these chaotic, turbulent flows. From this work with Roadrunner, we have learned a whole set of code design lessons that have served us well on other less unique machine designs. We believe that this experience will prove very valuable in finding ways to exploit future exascale computing systems.

References


