High-Performance Reactive Fluid Flow Simulations Using Adaptive Mesh Refinement on Thousands of Processors

A. C. Calder[1], B. C. Curtis[2], L. J. Dursi[1], B. Fryxell[1], G. Henry[3], P. MacNeice[4,5], K. Olson[1,5], P. Ricker[1], R. Rosner[1] F. X. Timmes[1], H. M. Tufo[1], J. W. Truran[1], M. Zingale[1]

ABSTRACT

We present simulations and performance results of nuclear burning fronts in supernovae on the largest domain and at the finest spatial resolution studied to date. These simulations were performed on the Intel ASCI-Red machine at Sandia National Laboratories using FLASH, a code developed at the Center for Astrophysical Thermonuclear Flashes at the University of Chicago. FLASH is a modular, adaptive mesh, parallel simulation code capable of handling compressible, reactive fluid flows in astrophysical environments. FLASH is written primarily in Fortran 90, uses the Message-Passing Interface library for inter-processor communication and portability, and employs the PARAMESH package to manage a block-structured adaptive mesh that places blocks only where resolution is required and tracks rapidly changing flow features, such as detonation fronts, with ease. We describe the key algorithms and their implementation as well as the optimizations required to achieve sustained performance of 238 GFLOPS on 6420 processors of ASCI-Red in 64 bit arithmetic.

1. Introduction

The origin and evolution of most of the chemical elements in the Universe are attributable to thermonuclear reactions in supernovae and novae. The FLASH code was designed to study thermonuclear flashes on the surfaces and in the interiors of stars. The code has been used to study a wide variety of astrophysical problems for almost two years, including helium burning on neutron stars (Zingale et al. 2000), laser driven shock fronts through multi-layer targets (Calder et al. 2000), turbulent flame fronts (Dursi et al. 2000), and carbon detonations in white dwarfs (Timmes et al. 2000).

[1]Center for Astrophysical Thermonuclear Flashes, The University of Chicago, Chicago, IL 60637

[2]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550

[3]Intel Corporation, Santa Clara, CA 92024

[4]Drexel University, Philadelphia, PA 19104-2875

[5]NASA Goddard Space Flight Center, Greenbelt, MD 20771

0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.

FLASH is a modular, adaptive mesh, parallel simulation code capable of handling compressible, reactive flows in astrophysical environments. It uses a customized version of the PARAMESH library (MacNeice et al. 2000) to manage a block-structured adaptive grid, placing resolution elements only where needed in order to track flow features. The compressible Euler equations are solved using an explicit, directionally split version of the piecewise-parabolic method (Colella & Woodward 1984), which allows for general equations of state using the method of Colella & Glaz (1985). An equation of state appropriate to stellar interiors is implemented using a thermodynamically consistent table lookup scheme (Timmes & Swesty 1999). Source terms for thermonuclear reactions are solved using a semi-implicit time integrator coupled to a sparse matrix solver (Timmes 1999). FLASH is implemented in Fortran 90 and uses the Message-Passing Interface library to achieve portability. Further details concerning the algorithms used in the code, the code's structure, and results of verification tests may be found in Fryxell et al. (2000).

The simulation described in this paper is of the propagation of a detonation front through stellar material. This particular problem was selected because it exercises most of the modules in the FLASH code, and thus gives a good representation of the performance of the entire code rather than a small subset of it. This problem is astrophysically important because it helps us determine how a supernova explodes, and thus aids in our understanding of the origin and evolution of the chemical elements. This simulation is also a computationally challenging because of the multiple length scales involved. A full supernova simulation would need to cover the 10^{-5} cm length scale of the burning front to the 10^9 cm scale of the star. In our model we focus on an intermediate range of spatial scales. The finest uniform meshes that can be run on the largest computers do not provide sufficient spatial resolution to answer some of the basic questions even at the intermediate range of length scales. By using adaptive mesh technology, the simulations described in this paper cover both the largest domain and the finest spatial resolutions ever conducted to date for this type of application.

In this paper we describe an efficient implementation of an adaptive mesh refinement package on massively parallel computers. Also covered is specific optimization and tuning on individual physics modules of the code in order to achieve good performance and scalability. We present performance results on ASCI Red using thousands of processors.

2. Adaptive Mesh Techniques on Parallel Architectures

Adaptive mesh refinement (AMR) offers the opportunity to make progress over uniform meshes by allowing for higher spatial resolution and/or larger problem domains. However, AMR introduces many of its own issues. Data location in memory changes dynamically as the flow evolves. The resulting frequent redistribution of data dramatically increases interprocessor communication. This redistribution must be done in such a way as to maintain load balancing and data locality. Also, in order to achieve sufficient adaptivity the data must be divided into small blocks, resulting in a larger surface to volume ratio, further increasing the communication costs. Finally, there is a

computational overhead in checking the refinement criteria. These issues need to be carefully considered and addressed to ensure the potential gain of AMR is not offset by its additional overhead, particularly on massively parallel computers.

There are a number of different approaches to AMR in the literature. Most AMR treatments have supported finite element models on unstructured meshes (e.g. Löhner 1987). These meshes have the advantage of being easily shaped to fit irregular boundary geometries, though this is not an issue for the vast majority of astrophysical problems. Unstructured meshes also require a large amount of indirect memory referencing, which can lead to relatively poor performance on cache-based architectures. Khokhlov (1997) has developed another strategy that refines individual Cartesian grid cells and manages the hierarchy as elements of a tree. This approach can produce very flexible adaptivity, in the same way that finite-element AMR does. It also avoids the guardcell overhead associated with the block structured approaches discussed below. However, the irregular memory referencing and expensive difference equation updates may be expected to produce slowly executing code. Khokhlov (1997) quotes a speedup for this code of roughly a factor of 15 in time to solution when compared to a uniform mesh code. However, no detailed performance numbers are given and the code was only run on a small number of processors (16 processors of a CRAY J90). We note that the speed increase of an AMR code over its uniform mesh counterpart is highly problem dependent.

PARAMESH falls into a third class known as block-structured AMR. Berger and co-workers (Berger 1982, Berger & Oliger 1984, Berger & Colella 1989) have pioneered these algorithms, using a hierarchy of logically Cartesian grids to cover the computational domain. This approach is flexible and memory-efficient, but the resulting code is complex and can be difficult to parallelize efficiently. Quirk (1991) and De Zeeuw & Powell (1993) implemented simplified versions which develop the hierarchy of nested grids by bisecting blocks in each coordinate direction and linking the hierarchy as the nodes of a data-tree.

Most of these AMR schemes have been developed within application codes to which they are tightly coupled. Some have been distributed as packages, with HAMR (Neeman 1999) and AMRCLAW (LeVeque & Berger 2000) being uniprocessor examples. We are currently aware of three publically available packages which support the block structured class of AMR on parallel machines; DAGH (Parashar 1999), AMR++ (Quinlan 1999), and SAMRAI (Kohn et al. 1999). All these packages use guardcells at block boundaries as a means of communicating information from surrounding blocks. Little performance information is available for these packages, and their scaling properties on thousands of processors has not, to our knowledge, been discussed. The DAGH, AMR++, SAMRAI, and PARAMESH packages are similar in many respects but do have some differences. Perhaps the most significant difference is that DAGH, AMR++, and SAMRAI are designed in terms of highly abstracted data and control structures. PARAMESH is designed with much less abstraction, resulting in a shorter learning curve and permitting a tight integration of PARAMESH with FLASH. These differences will have some impact on the execution speed of each package on a given parallel machine.

3. PARAMESH: Algorithms, Implementation, and Optimization

There are two critical data structures maintained with Fortran 90 in FLASH, one to store the model solution and the other to store the tree information describing the numerical grid. The solution data may be cell-centered or face-centered, although FLASH at present uses only cell-centered data. A hierarchy of 1D, 2D, or 3D blocks are built and managed with a tree structure (quad-tree in 2D or oct-tree in 3D).

Each block has $nxb \times nyb \times nzb$ cells and a prescribed number of guardcells at each boundary. The piecewise-parabolic method used by FLASH for the compressible hydrodynamics requires 4 guardcells at each boundary. These guardcells are filled with data from neighboring blocks or by evaluating the prescribed boundary conditions. The guardcell filling algorithm involves three steps. First, the data at "leaf" blocks are restricted (i.e. interpolated) to the parent block. Second, all blocks that are leaf blocks, or are parents of leaf blocks, exchange guardcell data with any neighboring blocks they might have at the same level of refinement. Third, guardcell data of the parents of leaf blocks is prolongated to any child block's guardcells that are at a jump in refinement. These steps involve interprocessor communication if the blocks which are exchanging data lie on different processors. The message is packed such that only the necessary data (using MPI derived data types) is exchanged between the sending and receiving blocks. Within a block structured form of AMR, these messages are quite large and easily overcome message latency concerns. For example, we typically have 24 variables per grid cell (including 13 nuclear isotopes) and 4 guardcells on each side of an $8 \times 8 \times 8$ block. This results in a message of size $24 \times 4 \times 8 \times 8 \times 8$ bytes $\sim$ 50 kbytes while filling guardcells with adjacent blocks having the same spatial resolution. Once all the guardcells of a block are filled, the solution on that block can be advanced independently of any other blocks.

Requiring that all grid blocks have an identical logical structure may, at first sight, seem inflexible and therefore inefficient. However, this approach has two advantages. First, the logical structure of the code is considerably simplified, which is a major concern in developing and maintaining robust parallel software. Second, the data-structures are defined at compile time, which gives modern compilers a better opportunity to manage cache use and extract superior performance.

Each block is refined by halving the block along each dimension and creating a set of new sub-blocks. Each sub-block has a resolution twice that of the parent block, and fits exactly in the borders of the parent block. Independent of any refinement criteria, there may be a jump of only one level of refinement across adjacent blocks. This restriction simplifies the data structures, and contributes to the stability of the conservative hydrodynamics algorithm operating on each block. For flux conservation at jumps in refinement, the mass, momentum, energy, and composition fluxes across the fine cell faces are added and provided to the corresponding coarse face on the neighboring block.

When a block is refined, its new child blocks are temporarily placed at the end of its processor's tree list. When a block is de-refined, sibling blocks are simply removed from the tree structure.

Fig. 1.— The Morton space filling curve for an arbitrary set of blocks of varying spatial resolution.

Memory locations of those blocks are overwritten by packing the list of remaining on-processor blocks. After all refinements and de-refinements are completed, the redistribution of blocks is performed using a Morton space-filling curve (Warren & Salmon 1987). Using a work-weighted Morton curve addresses the issues of load balancing and data locality. Such a Morton curve is shown in Figure 1. Other space-filling curves, which in principle have better spatial locality properties (Hilbert and Peano curves; Sagan 1994), have been tried in the FLASH code with no measurable improvement in performance. Additional possibilities such as graph algorithms or partial orderings have not yet been investigated.

The Morton number is computed by interleaving the bits of the integer coordinates of each block. The Morton-ordered list does not require a global sort. We have chosen our Morton ordering such that no pattern of refinement or de-refinement can result in a block with a Morton number lower than its parent nor larger than the Morton number of its neighbor. Thus, a global sort to restore a Morton ordered-list is unnecessary, and the re-distribution can be done entirely with local sorts and nearest-neighbor communications. Each block is then assigned a work value that can change as a function of time and/or position and is used to weight the blocks. The total list, in Morton order, is cut into a number of sublists equal to the number of processors, such that each sublist has roughly the same amount of work. Blocks of data are then moved to their new processors. This re-distribution step is only done if refinement or de-refinement occurs.

For high resolution simulations with time-dependent, complex features, refinement and de-refinement occur at each test. At one extreme, in a steady-state, planar detonation front the total number of blocks remains fairly constant; new blocks are created ahead of the front and removed behind the front. At the other extreme, in a spherical shock front the total number of blocks increases as the square of the radius. The number of blocks that are re-distributed is

highly dependent on the problem and the number of processors. If a lot of refinement is done in a small region then there is considerable re-distribution. If the blocks are relatively evenly spread throughout the volume, then there is little re-distribution. Our three-dimensional carbon detonation simulation features a planar front with an extended reaction zone, and on average transfers 65% of the mesh during the re-distribution step.

The criteria for refining a block are user-defined. We presently use a normalized second derivative (Löhner 1987). While the frequency of testing for refinement is an adjustable parameter, it is important to ensure that flow discontinuities are detected by a block before they move into the interior cells of that block. For production run simulations, we test for refinement every 4 timesteps. Examining the second derivative of the density, pressure and temperature is sufficient to guarantee an accurate solution to the detonation problem.

To organize the blocks across a multiprocessor machine, each block is stored at one of the nodes of a fully linked tree data structure. Each node stores its parent block, any child blocks it might have, and a list of its neighboring blocks. The list stores indices of abutting blocks at the same level of refinement in the north, south, east, west, up and down directions. If a neighboring block does not exist at the same level of refinement, a null value is stored. If a block is located at a boundary of the computational domain, the neighbor link in that direction is given a value to indicate the type of boundary condition. All the tree links of the children, parents and neighbors are stored as integers that indicate a local identification into a processor's memory. A second stored integer indicates on which processor that child, parent, or neighbor is located. The x, y, z coordinates and bounding box information for each block at each node in the tree are also stored.

An example of a two-dimensional domain and its tree structure is shown in Figure 2. The leaf blocks contain the current solution. The $8{\times}8$ interior cells within each block are also shown. A refined block has a cell size a factor of two smaller than its parent. The number near the center of each block is its Morton number. The symbols in the tree indicate on which processor the block would be located on a four-processor machine. As blocks are refined or de-refined, the tree links are reset accordingly.

For large numbers of blocks and processors, maintenance of the tree structure is potentially the most communications-intensive part of the FLASH code. The accumulated latency associated with the many small messages that must be sent to update the neighbor list creates a significant communications overhead, comparable to the overhead of the large messages sent in the redistribution of entire blocks. Some machine architectures ameliorate the worst effects of this overhead. For example, ASCI Red has low latency for small messages, and the interconnect speed is well balanced to the processor speed. Regardless, we pack messages whenever possible to reduce overhead.

Finally, an additional performance improvement came from removing as many explicit barrier commands as possible. PARAMESH was originally developed for the Cray T3E and took advantage of one-sided puts and gets of the SHMEM library. Explicit barrier commands, which are very inexpensive on the T3E, were required to ensure the safety of data transfers. Removing extraneous

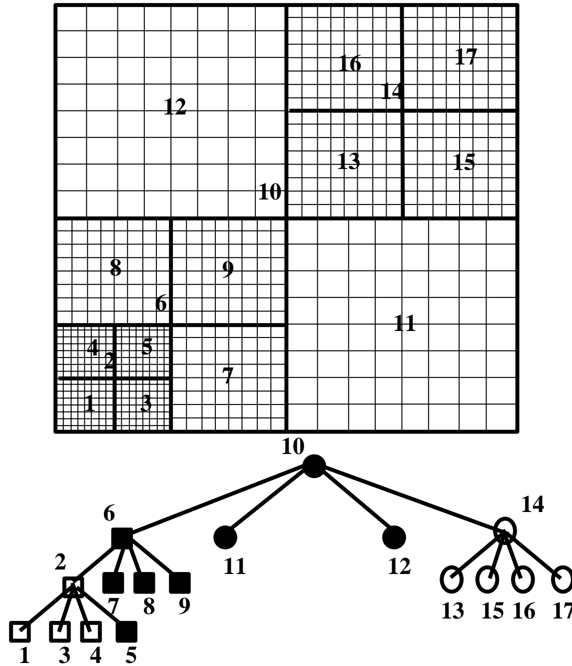

Fig. 2.— A simple set of blocks which cover a fixed domain.

barrier commands when porting PARAMESH from SHMEM to MPI improved the scaling by more than a factor of two on ASCI Blue Pacific with 512 processors. A larger number of processors was required on ASCI Red to achieve the same scaling improvement.

4. Tuning of Physics Kernels

The execution time of FLASH on the astrophysical problems of interest is dominated by computations of complicated local physics – the equation of state and nuclear reaction networks. In our carbon detonation calculations we require about 45,000 floating-point operations per cell per timestep, 95% of which are taken up by the physics kernels. The remaining 5% are taken up by the requisite interpolation for block creation/destruction and refinement checking. When setting up a problem, we select the smallest number of processors that allow the problem to fit in memory. For our carbon detonation simulations on ASCI Red, 60% of the execution time on 6400 processors is spent on computation; for fewer processors the computation to communication ratio is even larger. Thus, single processor optimizations can yield a significant performance increase. In addition, we also address communication costs by minimizing the surface area for guardcell exchanges, which is the largest interprocessor communication in FLASH. Near-perfect load balance is achieved by burning all of the blocks. As the number of blocks per processor increases, the computation to communication ratio also increases, decreasing the time to solution. Optimal performance is

achieved by running the calculation on the smallest number of processors possible for the memory requirement of the calculation.

Our first pass at optimization resides in the domain of single processor tuning. Often this meant replacing multiple divides by multiplications of the inverse, making as many array bounds known at compile time as possible, removing extraneous memory copies, and expressing operations involving a multiply followed by an add as a multiply-add instruction, since some of the ASCI platforms implement this instruction in hardware. All the arrays in FLASH were designed to have unit stride loop indices and have the first index be the loop index in order to achieve good cache performance with Fortran. Vendor supplied libraries for exponentials, logarithms, and power functions were invoked when available. The loss of accuracy in using these platform-tuned libraries is typically in the last bit, which is not significant for our calculations. Tuning of the physics kernals increased the single processor performance from 30 MFLOPS to 60 MFLOPS on ASCI Red.

The equation of state is the most time consuming routine in our carbon detonation simulations, accounting for $\sim 25\%$ of the total run time. This expense is primarily due to the large number of times the equation of state routine is called, either for ensuring consistency of the hydrodynamics with the thermodynamics, maintaining thermodynamic consistency in the guardcells, or updating the thermodynamic state after a thermonuclear burn (or any other energy source term) has been called. The stellar equation of state implements a thermodynamically consistent table lookup scheme (Timmes & Swesty 1999). There are nine hashed, rectangular tables which consume a total of ~ 1 Mbyte of memory. Each table is two-dimensional and stores the Helmholtz free energy or one of its derivatives as a function of temperature and density. Simple scaling laws account for differences in composition and avoid the use of many three-dimensional tables. Since the reactive fluid equations evolve the density and internal energy, a Newton iteration is required to find the temperature. Storing the tables as a function of density and internal energy is not desirable, as the tables cannot be hashed and the simple compositional scaling laws would be lost. Finally, our stellar equation of state routine is vectorized, which boosts performance by taking advantage of the large CPU to L1/L2 cache bandwidth. The longer the vector, the larger the performance boost. Wherever possible, we map a block of data into a one-dimensional vector and execute the equation of state over the entire block. Our implementation of the equation of state is the most efficient we are aware of (97 MFLOPS on a single processor of ASCI Red), yet yields the accuracy that we require (Timmes & Arnett 1999). Our single processor tuning of the equation of state decreased the execution time by 20% when called for a $8 \times 8 \times 8$ block.

Integration of the stiff ordinary differential equations that represent the nuclear reaction network is comparable in cost to the equation of state. This cost is primarily due to evaluating 110 analytical nuclear reaction rates, which contain numerous exponential and power function calls, and assembling the complicated right hand sides of the differential equations. Evaluating the reaction rates for the 13 isotopes in the reaction network is twice the cost of successfully integrating the ordinary differential equations with a variable order, semi-implicit method (Timmes 1999). We used the GIFT program for the linear algebra portion of the integration. GIFT is a program which generates

Fortran subroutines for solving a system of linear equations by Gaussian elimination (Gustafson, Liniger, & Wiloughby 1970; Müller 1997). GIFT-generated routines skip all calculations with matrix elements that are zero; in this sense GIFT-generated routines are sparse, but the storage of a full matrix is still required. GIFT assumes diagonal dominance of the pivot elements and performs no row or column interchanges. GIFT writes out (in Fortran) the sequence of Gaussian elimination and backsubstitution without any do loop constructions over the matrix elements. As a result, the routines generated by GIFT can be quite large. However, for small matrices the execution speed of GIFT-generated routines on nuclear reaction networks is faster than all dense or sparse packages that we tested (Timmes 1999). The GIFT-generated routines were hand optimized, by removing unnecessary divides, to achieve better single processor performance. Our single processor tuning of the nuclear reaction networks decreased the execution time by 30% when called for a single cell.

The third most expensive module in this calculation is the Eulerian hydrodynamic solver, which consumes $\sim$ 20% of the total run time. This expense is mainly due to the complexity of the piecewise-parabolic method. While expensive, the method allows accurate solutions with fewer grid points, which improves the time to solution. The hydrodynamics module operates on one-dimensional vectors with unit stride for optimal memory access and cache performance. Good performance is also insured by using blocks small enough to fit entirely inside cache, and accessing the PARAMESH data structures directly to eliminate unnecessary, expensive memory copies.

5. Performance of the Code

Our benchmark carbon detonation calculations were performed on ASCI Red, which consists of 3212 nodes in its large configuration, each of which contains two 333 MHz Intel Pentium II processors with Xenon Core technology and 256 Mbytes of shared memory. Performance was measured using the perfmon library which access the hardware counters. Hardware counters were flushed at the start of each timestep and read at the end. We report on the performance for 64 bit arithmetic because the range of magnitudes spanned by the energies and reaction rates exceeds the range of 32 bit arithmetic.

Our benchmark calculation is a three-dimensional simulation of a burning front propagating through stellar material. This simulation has importance for supernova models, since it may significantly affect the evolution of the chemical elements produced in the explosion. The size of the computational domain of the simulations we present is 12.8 $\times$ 256.0 cm in two dimensions, and 12.8 $\times$ 12.8 $\times$ 256.0 cm in three dimensions. We varied the maximum spatial resolution from 0.1 to 0.0125 cm.

Although interesting physics occurs in only a small portion of the domain at any given time, having a large domain is necessary for obtaining reliable answers. For example, there is a natural evolution from the initial conditions to the steady state. Other possible approaches to modeling this application suffer from various numerical errors. A moving reference frame with a stationary

detonation front is prone to well-known errors associated with slowly-moving shocks. Having a constant number of cells such that grid is added ahead of the detonation front and removed behind the front suffer from not satisfying the boundary conditions. The desirability of using a large domain makes this application a prime candidate to benefit from an efficient implementation of AMR.

These carbon detonation calculations on a uniform mesh would be prohibitively expensive in terms of the required CPU time, memory, and disk space. The finest resolution would require 1024 $\times$ 20,480 $\sim$ 21 million cells in two dimensions. In three dimensions with 64 bit arithmetic, this resolution would require 21 billion cells, 4 Tbytes of memory, and 3 node-years per time step. An entire calculation would require 3 years on all 6424 processors of ASCI Red, if ASCI Red had sufficient memory. With AMR, only $\sim$ 2 million cells are needed in two dimensions, for a factor of 10 savings in the number of grid points. In three dimensions $\sim$ 512 million cells would be required, for a factor of 40 savings.

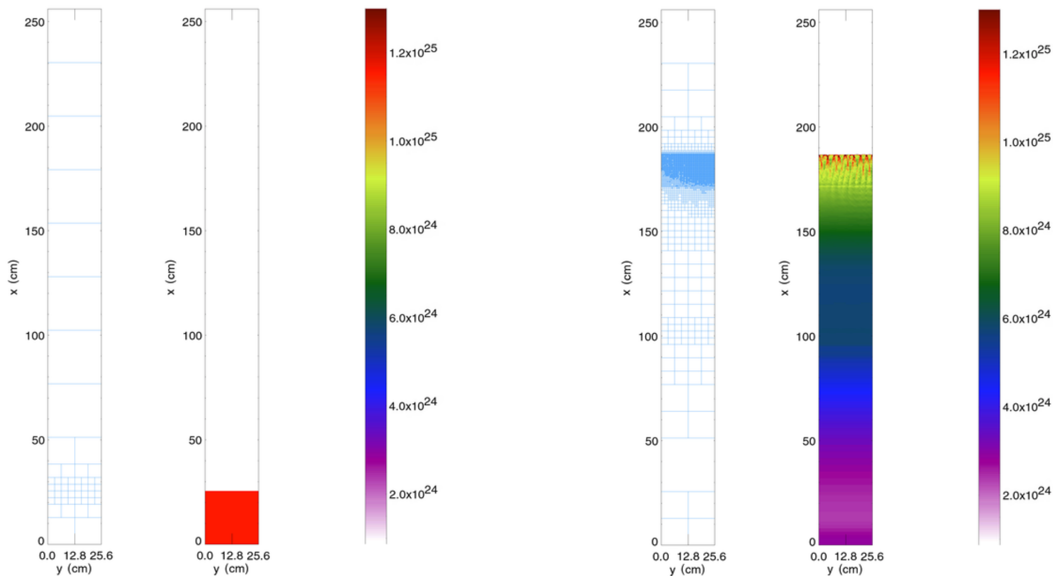

Fig. 3.— Pressure field and computation mesh for the initial conditions and after $1.26{\times}10^{-7}$ s.

Figure 3 shows the pressure field of a two-dimensional carbon detonation at the initial time and after $1.26{\times}10^{-7}$ s. The interacting transverse wave structures are particularly vivid at $1.26{\times}10^{-7}$ s, and extend about 25 cm behind the shock front. The figure also shows the block structure of the mesh, with each block containing 8 grid points in the x- and y-directions. The entire mesh is rather coarse at the initial time; only the shocked interface is refined. At $1.26{\times}10^{-7}$ s the finest grids are placed only where there is significant structure in the thermodynamic variables. Thus, the mesh is not refined ahead of the detonation front where it is not needed, maximally refined throughout the reactive regions, and de-refined behind the detonation front as the fluid flow becomes smooth.

Figure 4 shows the silicon ashes in the maximally refined region of Figure 3 for two different spatial resolutions. There are appreciable variations in the spatial distribution of the ashes, and a sharp visual contrast between the two spatial resolutions. The image on the left is for a maximum spatial resolution of 0.1 cm (5 levels of refinement), and is equivalent to the finest resolution previously achieved on a uniform mesh (Gamezo et al. 1999). The image on the right is for a maximum spatial resolution of 0.0125 cm (8 levels of refinement), and represents the finest resolution to date for this application by a factor of 8 in each direction.

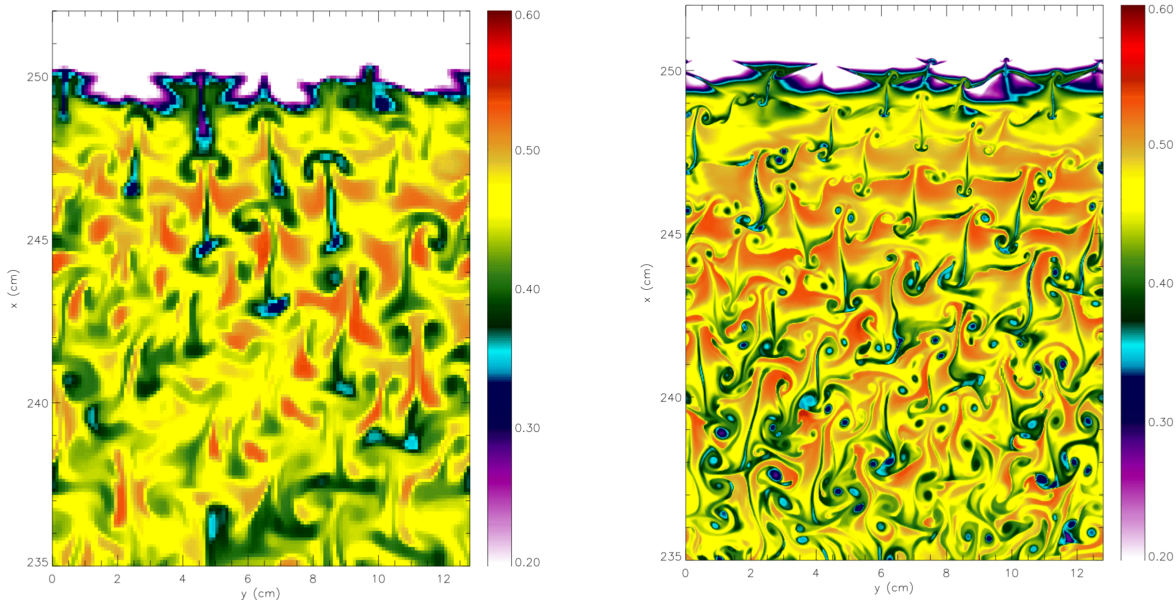

Fig. 4.— Silicon abundances (the ashes) at two different spatial resolutions.

Figure 5 shows the three-dimensional pressure field of a carbon detonation. In the image on the left, low pressure regions are blue, high pressure regions are red, and the dimpled surface of the detonation front is colored brown. On the volume rendered image on the right, high pressure regions are accented in red, while while the low pressure regions are suppressed and shown as shades of blue. The pressure variations in both images are caused by interacting transverse wave structures. The spatial resolutions of these three-dimensional images is 0.1 cm, and represents the finest resolution to date for this application in three dimensions.

Table 1 summarizes the performance characteristics of the three-dimensional cellular detonation problem on ASCI Red. The single processor optimizations described in the previous sections resulted in a factor of two improvement in time to solution. The parallel processor tuning described above yielded a factor of two improvement in scaling out to large numbers of processors. For each benchmarking run, the table lists the number of processors, number of AMR levels, number of computational blocks on each processor, average MFLOPS per processor, and the aggregate GFLOPS. Each benchmark run was typically run for 50-100 timesteps. As noted above, ASCI Red has two

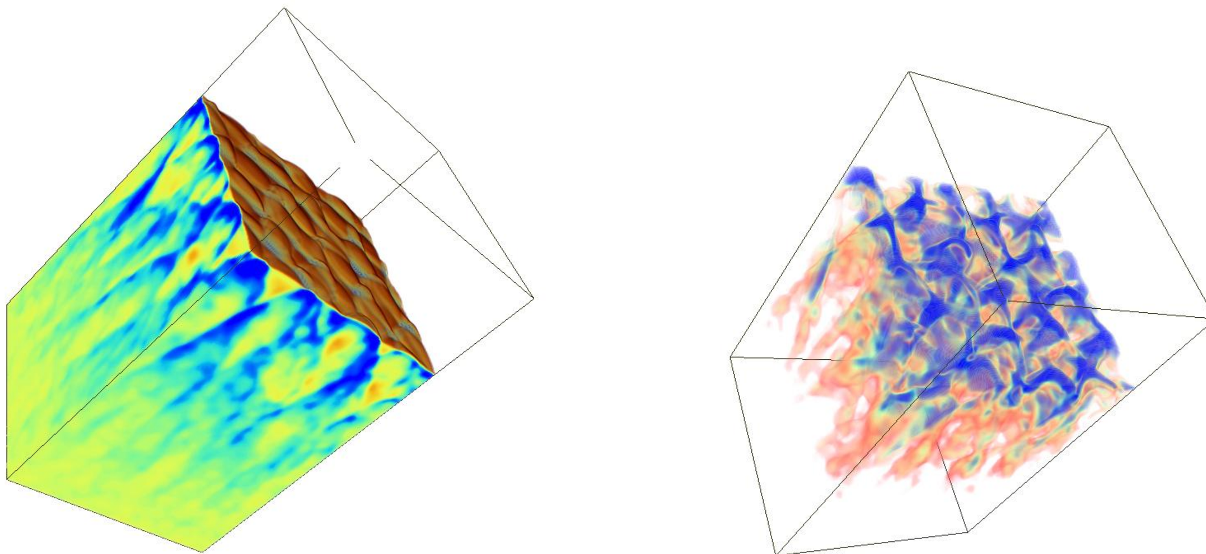

Fig. 5.— Pressure fields in three-dimensional carbon detonation.

processors per node. The upper portion of Table 1 shows the results when both processors are used for MPI tasks (proc-3 mode), while the lower portion shows the result when only one processor is used for MPI tasks (proc-0 mode). One expects lower performance in proc-3 mode because each processor in a node accesses the same pool of memory, which effectively halves the memory bandwidth available to each processor. This decrease in performance is reflected in the table. The table shows that as the number of processors increases from 4096 to 6420, for the same size of problem, we maintain a 94% scaling efficiency as measured by MFLOPS.

Table 1 also shows that the performance increases with a larger number of blocks per processor. Increasing the number of blocks increases the amount of computation relative to the communication cost. We used up to 70% of the available 256 Mbytes of memory on each node. On the largest run, we achieved 37.0 MFLOPS per processor, reaching an aggregate of 238 GFLOPS on 6420 processors in 64 bit arithmetic, 11% of the theoretical peak.

6. Conclusions

We have presented performance results for the largest simulations of nuclear burning fronts in supernovae conducted to date, which we have performed using the FLASH adaptive-mesh code on the Intel ASCI-Red machine at Sandia National Laboratories. We have described the key algorithms and their implementation as well as the optimizations required to achieve sustained performance of 238 GFLOPS on 6420 processors of ASCI-Red in 64 bit arithmetic.

Table 1: Performance Statistics on ASCI Red

Number of Processors	Refinement Levels	Blocks per Processor	MFLOPS per Processor	Total GFLOPS
Two Processors per Node				
6420	8	33.1	37.0	238
4096	8	36.1	39.2	161
2048	7	33.7	40.2	82
One Processor per Node				
3072	8	58.2	47.4	146
2048	8	72.1	45.2	93
1024	7	63.5	50.2	51

Adaptive-mesh simulations pose many complex design problems, and a variety of different techniques exist to solve these problems. Several packages use local-cell or block-structured refinement, both with and without temporal refinement. The implementation of the PARAMESH library that is used by FLASH is a block-structured AMR package which evolves all blocks on the same timestep. This lack of temporal adaptivity simplifies our time integration and load-balancing but prevents us from taking best advantage of AMR for highly refined meshes. A further simplification has been possible because we do not need to solve a Poisson equation for hydrodynamics; the problem we are solving is highly compressible. However, we are currently developing a multigrid-based and two-level (Tufo & Fischer 1999, 2000) elliptic solvers for use with future self-gravitating problems to be performed with FLASH. With the complex local physics included in our calculations, load-balancing and communication costs are less important than they might be in simpler fluid problems. However, including temporal refinement and elliptic solvers will raise new load-balancing issues which we expect to address in future work.

The problem we have described in this paper represents the direct numerical simulation of microscopic physical processes important to a large-scale problem, that of supernova explosions. Adaptive-mesh techniques have been essential to these calculations. However, the important length scales in a supernova explosion range from less than 10^{-5} cm to more than 10^{9} cm – a range of scales which cannot be addressed with AMR methods alone. Thus in performing large-scale simulations of supernovae, we need to include parameterized subgrid models of small-scale phenomena like nuclear burning fronts. These phenomena will also require some type of front-tracking technique. Direct numerical simulations like the ones presented here are important for calibrating these subgrid models. Subgrid models should make large-scale supernova calculations feasible in the near future.

In summary, adaptive mesh refinement has permitted us to address problems of astrophysical

interest which we could not otherwise afford to solve. Because these problems are dominated by local physics, traditional single-processor optimizations have allowed us to obtain improvements of almost a factor of two over previous versions of our code. More complex communication-related optimizations, such as using the Morton curve to allocate blocks to processors, have also improved performance. Throughout our optimization work we maintained portability.

This work is supported by the Department of Energy under Grant No. B341495 to the Center for Astrophysical Thermonuclear Flashes at the University of Chicago, and under the NASA HPCC/ESS project. K. Olson acknowledges partial support from NASA grant NAS5-28524, and P. MacNeice acknowledges support from NASA grant NAS5-6029.

REFERENCES

Berger, M.J. 1982, Ph.D. Thesis, Stanford Univ.

Berger, M.J. & Oliger, J. 1984, J. Comp. Phys., 53, 484

Berger, M.J. & Collela, P. 1989, J. Comp. Phys., 82, 64

Calder, A.C., Fryxell, B., Rosner, R., Kane, J., Remington, B.A., Dursi, L.J., Olson, K., Ricker, P.M., Timmes, F.X., Zingale, M., MacNeice, P., & Tufo, H. 2000, BAAS, 32, 704

Colella, P. & Glaz, H. M. 1985, J. Comp. Phys., 59, 264

Colella, P.& Woodward, P. 1984, J. Comp. Phys., 54, 174

DeZeeuw, D. & Powell, K. G. 1993, J. Comp. Phys., 104, 56

Dursi, J., Niemeyer, J., Calder, A., Fryxell, B., Lamb, D., Olson, K., Ricker, P., Rosner, R., Timmes, F.X., Tufo, H., & Zingale, M. 2000, BAAS, 31, 1430

Fryxell, B.A. Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., & Tufo, H. 2000, Astrophysical J. Supp., in press

Gustafson, F.G., Liniger, W., & Wiloughby, R. 1970, J. Assoc. Comput. Mach., 17, 87

Gamezo, V.N., Wheeler, J.C., Khokhlov, A.M., & Oran, E.S. 1999, Astrophysical J., 512, 827

Khokhlov, A. M. 1997, Memo 6406-97-7950, (Naval Research Lab)

LeVeque, R. & Berger, M. 2000, http://www.amath.washington.edu:80/~rjl/amrclaw/

Löhner, R. 1987, Comp. Meth. App. Mech. Eng., 61, 323

MacNeice, P., Olson, K.M., Mobarry, C., de Fainchtein, R., & Packer, C. 2000, Comp. Phys. Comm., 126, 330

Müller, E. 1997, Saas-Fee Advanced Course 27, eds. O. Steiner & A. Gautschy (Berlin: Springer), p.343

Neeman, H. J. 1996, Ph.D. Thesis, Univ. of Illinois

Parashar, M. & Browne, J.C. 2000, in IMA Volume 117: Structured Adaptive Mesh Refinement Grid Methods, eds. S.B. Baden, N.P. Chrisochoides, D.B. Gannon, & M.L. Norman (Berlin: Springer-Verlag)

Quinlan, J.J. 1991, Ph.D. Thesis, Cranfield Inst. Tech.

Sagan, H. 1994, Space-Filling Curves (Berlin: Springer-Verlag)

Timmes, F.X. 1999, Astrophysical J. Supp., 124, 241

Timmes, F.X., Zingale, M., Olson, K., Fryxell, B., Ricker, P., Calder, A.C., Dursi, L.J., Tufo, H., MacNeice, P., Truran, J.W., & Rosner, R. 2000, Astrophysical J. Supp., in press

Timmes, F.X., & Arnett, D. 1999, Astrophysical J. Supp., 125, 294

Timmes, F.X., & Swesty, F.D. 2000, Astrophysical J. Supp., 126, 501

Tufo, H.M., & Fischer, P.F. 1999, Proc. Supercomputing 1999 (Portland, OR: IEEE Computer Soc.)

Tufo, H.M., & Fischer, P.F. 2000, J. Par. & Dist. Computing, in press

Warren, M.S. & Salmon, J.K. 1993, Proc. Supercomputing 1993 (Washington, D.C.: IEEE Computer Soc.), p.12

Zingale, M., Timmes, F.X., Fryxell, B., Lamb, D.Q., Olson, K., Calder, A.C., Dursi, L.J., Ricker, P., Rosner, R., MacNeice, P., & Tufo, H. 2000, Astrophysical J., in press

This preprint was prepared with the AAS LaTeX macros v5.0.