NAS Technical Report
NAS-06-003
May 2006

Performance Characteristics of an Adaptive Mesh Refinement Calculation on Scalar and Vector Platforms

Michael Welcome, Charles Rendleman, Leonid Oliker
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
{mlwelcome,carendleman,loliker}@lbl.gov

Rupak Biswas
NASA Advanced Supercomputing Division
NASA Ames Research Center
Moffett Field, CA 94035
rbiswas@mail.arc.nasa.gov

ABSTRACT

Adaptive mesh refinement (AMR) is a powerful technique that reduces the resources necessary to solve otherwise intractable problems in computational science. The AMR strategy solves the problem on a relatively coarse grid, and dynamically refines it in regions requiring higher resolution. However, AMR codes tend to be far more complicated than their uniform grid counterparts due to the software infrastructure necessary to dynamically manage the hierarchical grid framework. Despite this complexity, it is generally believed that future multi-scale applications will increasingly rely on adaptive methods to study problems at unprecedented scale and resolution. Recently, a new generation of parallel-vector architectures have become available that promise to achieve extremely high sustained performance for a wide range of applications, and are the foundation of many leadership-class computing systems worldwide. It is therefore imperative to understand the tradeoffs between conventional scalar and parallel-vector platforms for solving AMR-based calculations. In this paper, we examine the HyperCLaw AMR framework to compare and contrast performance on the Cray X1E, IBM Power3 and Power5, and SGI Altix. To the best of our knowledge, this is the first work that investigates and characterizes the performance of an AMR calculation on modern parallel-vector systems.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance Attributes; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Performance, Experimentation, Measurement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'06, May 3–5, 2006, Ischia, Italy.
Copyright 2006 ACM 1-59593-302-6/06/0005 ...$5.00.

Keywords

HyperCLaw framework, high end computing, Cray X1E, SGI Altix, IBM Power3 and Power4, integrated performance monitoring

1. INTRODUCTION

Adaptive mesh refinement (AMR) is a powerful technique that reduces the computational and memory resources required to solve otherwise intractable problems in computational science. Typically, AMR has been applied to physical systems that are modeled by a governing set of partial differential equations (PDEs). The AMR strategy solves the system of PDEs on a relatively coarse grid, and dynamically refines it in regions of scientific interest or where the coarse grid error is too high for proper numerical resolution. Without some form of adaptivity, naively increasing the grid resolution uniformly across the entire computational domain can become prohibitively expensive for realistic cases. For example, in computational fluid dynamics (CFD), the size of a stable timestep is limited by the Courant-Friedrichs-Levy (CFL) condition (i.e. if the grid spacing decreases by a factor of 2, the timestep must also be decremented by a factor of 2). For a 3D problem, halving the mesh spacing in each direction increases memory requirements by 8x and the computational work required to advance the solution to the same point in time by 16x. Adaptive methods operate by restricting this refinement to a relatively small portion of the computational domain.

However, the price paid for this additional power is complexity. Adaptive codes tend to be far more complicated than their uniform grid counterparts. Significant software infrastructure must be developed for the dynamic management of refined regions and sophisticated numerical techniques must be used to ensure that the PDEs are satisfied at the interface between coarse and refined grids. This includes mechanisms to accurately interpolate and coarsen data between refined and unrefined regions. Because of this additional overhead, AMR codes tend to be built over frameworks or libraries that provide the desired functionality. They are also usually written in a flexible and modular manner to maximize re-use and/or simplify inclusion of alternate physics. This, in turn, may result in lower comparative performance than uniform mesh codes, even when run with adaptivity turned off.

Despite this complexity and the growth of modern high end computing (HEC) systems to 10,000's of processors, it is

generally believed that the use of adaptive methods will continue to grow as calculations involving multi-scale physics become more feasible and prevalent. Scientists will continue to increase problem sizes and resolutions that will remain computationally intractable without adaptive methods. Examples of AMR frameworks include BoxLib/AmrLib [4, 15] developed by the Center for Computational Sciences and Engineering (CCSE) at Lawrence Berkeley National Laboratory (LBNL), Chombo [5] developed by ANAG also at LBNL, FLASH [9] from the University of Chicago, SAMRAI [16] from Lawrence Livermore National Laboratory, and Grace/DAGH [14] developed at Rutgers University.

Recently, a new generation of parallel-vector architectures have become available to the supercomputing community. These systems promise to achieve extremely high sustained performance for a wide range of applications, and are the foundation of many leadership-class computing systems worldwide. It is therefore imperative to understand the tradeoffs between conventional scalar and parallel-vector platforms for solving AMR-based calculations. In this paper, we examine the HyperCLaw AMR framework developed at LBNL to critically compare and contrast performance on the Cray X1E, the IBM Power3 and Power5, and the SGI Altix. To the best of our knowledge, this is the first work that investigates and characterizes the performance of an AMR calculation on modern parallel-vector systems and compares it against that obtained on state-of-the-art scalar platforms.

2. ADAPTIVE MESH REFINEMENT CALCULATION

In this section, we present some basic concepts of adaptive mesh refinement (AMR), describe the HyperCLaw AMR library, discuss the key AMR components of our calculation, and highlight the major parallelization and vectorization issues.

2.1 Methodology

AMR is a popular and powerful technology for solving partial differential equations (PDEs) using a hierarchy of grids of differing resolution ranging from the coarsest to the finest. Each refinement level is represented as a union of rectangular grid patches of a specific resolution contained within the computational domain. In this work, the level 0 grid is a single rectangular 3D parallelopiped constituting the problem domain. The refinement factor, r, is the ratio in resolution between consecutive levels, and is uniform in all spatial directions. The grids are properly nested, i.e. the union of grids at level $l+1$ is contained in the union of grids at level l. The containment is strict in the sense that, except at physical boundaries, the level l grids are large enough to ensure that there is a border at least one level l cell surrounding each level $l+1$ grid.

Both the initial creation of the grid hierarchy and the subsequent regridding operations use the same set of procedures to create new grids. Regridding is the process of dynamically changing the grid hierarchy to adequately capture physical phenonema of interest. Cells requiring enhanced resolution are identified and tagged using a user-supplied error indicator, and the tagged cells are grouped into rectangular patches. The new patches generally contain some cells that were not tagged for refinement. These rectangular patches are subdivided to form the grids at the next level. The process is repeated until either the error tolerance criteria is satisfied or a specified maximum level of refinement is reached. When new grids are created at level $l+1$, the data on these grids are copied from previous grids at the same level (where possible) using an efficient point-to-point protocol; otherwise the data is interpolated from the underlying level l grids.

The PDEs at a given level in the grid hierarchy are solved using Dirichlet data obtained from coarser levels. This results in flux errors at the boundary with the coarse grid, which are then corrected in a synchronization step when the coarse and fine grid solutions reach the same time. Boundary data is provided by filling ghost cells in a band around the fine grid data. The width of the band is determined by the stencil of the finite difference scheme. The time-stepping algorithm recursively advances grids at different levels using timesteps appropriate to that level based on CFL considerations, while the flux corrections are typically imposed in a time-averaged sense.

When the coarse and fine grid solutions reach the same time and are synchronized, two corrections need to be made. First, for all coarse grid cells covered by finer cells, the coarse data is replaced by the volume-weighted average of the fine grid data. Second, because coarse cells adjacent to the fine cells were advanced using different fluxes than those used for the fine cells, the coarse cell values are corrected by adding the difference between the coarse and fine grid fluxes.

2.2 HyperCLaw Library

HyperCLaw is a hybrid C++/`Fortran` AMR code developed and maintained by CCSE at LBNL [4, 15] where it is frequently used to solve systems of hyperbolic conservation laws. The PDEs are solved using a higher-order Godunov method that has evolved from and is based on the original method of Colella [6]. The base library, known as Boxlib, provides C++ classes and data containers for representing block-structured data and software for distributing/exchanging data on parallel computers using MPI. A second library, AmrLib, implemented in BoxLib, supports AMR methods on block-structured data. The algorithms contained in AmrLib are much the same as those described in the original AMR papers [2, 3]. Techniques to parallelize AMR methods within the BoxLib framework are described in [15]. These libraries, in addition to the tasks of controlling the details of a calculation (number of levels of refinement, number of timesteps between adaptations, parallel I/O, etc.), implement the main functionality of an AMR method.

The remainder of the HyperCLaw code consists of an applications layer containing the physics classes defined in terms of virtual functions within the AmrLib class hierarchy. The basic idea is that data blocks are managed in C++ in which ghost cells are filled and temporary storage is dynamically allocated so that when the calls to the physics algorithms (usually finite difference methods implemented in `Fortran`) are made, the same stencil can be used for all points and no special treatments are required. By structuring the software in this manner, the high level objects that encapsulate the functionality for AMR and its parallelization are independent of the details of the physics algorithms and the problem being solved. This simplifies the process of adding/replacing physics modules as long as they adhere to the AmrLib interface requirements.

```
Recursive Procedure Advance (level l)
    if Regrid needed at level l + 1
        Estimate errors at level l + 1
        Generate new grids for levels l + 1 to MRL
        Generate parallel data distributions
            using Knapsack
        Fill new grid data using interpolation
    endif
    Execute TimeStep by:
        if (l == 0) obtain boundary data from
            physical boundary conditions
        else Obtain boundary data from coarser grids
        Integrate level l in time using Godunov
        if (l < MRL)
            repeat r times: Advance (level l + 1)
        endif
        Synchronize data between levels l and l + 1
End Recursive Procedure Advance
```

Figure 1: Pseudo-code of the basic AMR algorithm.

2.3 Major AMR Modules

For the purposes of this discussion, the AMR algorithm can be described by the pseudo-code shown in Figure 1. Basically, the ***Advance*** procedure recursively advances refinement level l, $0 \leq l \leq$ MRL (maximum refinement level), with r invocations at each level (r is the refinement ratio). The repetition is required to comply with the CFL constraint: because the resolution at level $l+1$ is r times that of level l, the timestep must be cycled with stepsize r times smaller.

We now briefly describe each of the major modules of the HyperCLaw framework.

Godunov:

This is the phase where the majority of the computational work is performed by advancing the solution on uniform grids at level l. It involves conservatively converting state variables to primitive form, computing 4th-order slopes in all directions, characteristic tracing of all variables, solving the Riemann problem to compute edge fluxes, and finally updating and converting back to state variables [6]. It is a computationally intensive phase, requiring approximately 1000 flops per cell. The `Fortran` routines operate on 3D arrays containing the discretized state of the physical system. There is no inter-processor communication.

TimeStep:

This phase prepares the grids at level l for the Godunov solver. This includes allocation of temporary storage, filling the ghost zones surrounding each grid patch, initializing and updating edge-based flux registers for the follow-on synchronization step, and computing a global CFL number. Communication is required to obtain initial data for the ghost cells either by copying from other grids at the same level (but possibly on other processors) or by spatial and temporal interpolation from coarse levels.

Regrid:

The function of this module is to replace an existing grid hierarchy with a new hierarchy in order to maintain numerical accuracy as important solution features develop and move through the computational domain. This includes tagging coarse cells for refinement, and buffering them to ensure that neighboring cells are also refined. The list of tagged cells are collected by a master processor and then re-broadcast. Each processor then constructs grid boxes to cover the tagged cells and maps the patches to processors to balance the computational workload. The algorithm is deterministic so that each processor generates the same grids and mappings without any communication. Each processor then allocates memory and initializes data for the grids that it owns. The procedure works down from finest level to ensure that grids at level l properly contain the projection of the level $l+1$ grids. This is primarily a serial algorithm performing integer operations on arrays of tagged cells and grid boxes.

Knapsack:

A new data distribution is performed during this phase. Data for the newly-generated fine grids is obtained by either copying from locations where there is existing fine grid data, or by interpolating in space from locations overlaid by data at coarser levels. This algorithm scales poorly with the number of grids as it is inherently sequential.

2.4 Parallelization and Vectorization

HyperCLaw required only minor effort to vectorize and multi-stream the `Fortran` loops on the X1E. Performance could have been improved had we restructured the Godunov solver to be more cache friendly; however, the original code was designed for flexibility and modularity that we did not want to compromise. The organization is logical and proceeds in phases: first compute 4th-order slopes, then characteristic tracing, then Riemann solves, and finally compute fluxes and update state variables. The net effect on performance is to load values into cache, perform a few floating-point operations, and then write back to memory. But many of the same values are reloaded into cache during a later phase of the algorithm. Rewriting the code by merging distinct phases would require substantial effort to enhance cache reuse, and make the code more monolithic and less flexible. Furthermore, our goal was less focused on trying to optimize the AMR calculation for a particular machine than understanding its performance properties on parallel-vector and scalar architectures.

HyperCLaw is based on a coarse-grained message-passing model using MPI as the parallelization strategy. In this paradigm, the grids or data blocks at a given level are distributed across processors so as to balance computational workload and minimize inter-processor communication. For typical AMR calculations, it is extremely difficult to achieve perfect load balance because of the wide range of block sizes. (The block size is the total number of cells in the block.) Our current implementation, based on a dynamic programming approach due to Crutchfield [8, 15], is very general in terms of the range of block sizes that can be handled. Generality however leads to additional overhead. On each processor, we duplicate the mapping between processors and the array of grids it contains. This provides excellent optimization for pre-computing the communications required for filling boundary data. The cost of this method increases quickly with the number of blocks, but is usually negligible for up to 1000 blocks.

Name	Platform	P/ Node	Clock (MHz)	Peak (GF/s/P)	Stream BW (GB/s/P)	Ratio (Byte/Flop)	MPI Lat (μsec)	MPI BW (GB/s/P)	Network Topology
Seaborg	Power3	16	375	1.5	0.4	0.26	16.3	0.13	Fat-tree
Bassi	Power5	8	1900	7.6	6.8	0.85	4.7	1.1	Fat-tree
Columbia	Altix	2	1500	6.0	2.0	0.33	2.8	1.0	Fat-tree
Phoenix	X1E	4	1130	18.0[1]	9.7	0.54	5.0	2.9	4D-Hcube

Table 1: Architectural highlights of Seaborg, Columbia, and Phoenix HEC platforms.

3. TARGET ARCHITECTURES

We briefly describe the salient features of the four diverse HEC platforms in our study. Table 1 presents an architectural overview of the Seaborg, Bassi, Columbia, and Phoenix systems, including: STREAM benchmark results [17] showing the measured EP-STREAM [11] triad bandwidth when all processors within a node simultaneously compete for main memory; the ratio of STREAM bandwidth to the peak computational rate; the measured internode MPI latency [13]; and the measured bidirectional MPI bandwidth per processor pair when each processor simultaneously exchanges data with a distinct processor in another node.

3.1 Seaborg (IBM Power3)

The Power3 was first introduced in 1998 as part of IBM's RS/6000 series. Each 375 MHz processor contains two floating-point units (FPUs) that can issue a multiply-add (MADD) per cycle for a peak performance of 1.5 Gflop/s. The Power3 has a pipeline of only three cycles, thus using the registers very efficiently and diminishing the penalty for mispredicted branches. The out-of-order architecture uses prefetching to reduce pipeline stalls due to cache misses. The CPU has a 32KB instruction cache, a 128KB 128-way set associative L1 data cache, and an 8MB four-way set associative L2 cache with its own private bus. Each SMP node consists of 16 processors connected to main memory via a crossbar. Multi-node configurations are networked via the Colony switch using an omega-type topology. The Power3 experiments reported here were conducted on Seaborg, the 380-node system running AIX 5.1 and operated by LBNL.

3.2 Bassi (IBM Power5)

The latest processor in the IBM Power line, the Power5 processor uses a RISC instruction set with SIMD extensions. The 1.9 GHz CPU contains a 64KB instruction cache, a 1.9MB on-chip 10-way set associative L2 cache as well as a massive 36MB on-chip 12-way set associative L3 cache (that is not part of the core). The IBM custom System Interface Chip (SMI) allows memory bandwidth to be aggregated across four DDR 233 MHz channels. Each Power5 chip has two SMI interfaces allowing an impressive measured STREAM performance (with respect to conventional DDR memory subsystems) of 6.8GB/s per processor. The peak floating-point performance of the evaluated Power5 system is 7.6 GFlop/s (two MADDs per cycle). The Power5 includes an integrated memory controller, previously an off-chip component, and integrates the distributed switch fabric between the memory controller and the core/caches. Each SMP node consists of eight processors, and is interconnected via a two-link network adaptor to the IBM Federation HPS switch. The Power5 experiments reported here were conducted on Bassi, the 122-node system running AIX 5.2 and operated by LBNL.

3.3 Columbia (SGI Altix)

Introduced in early 2003, the SGI Altix systems are an adaptation of the Origin 3000, which use SGI's NUMAflex global shared-memory architecture. This design enables the major subsystems to be packaged into modular components, called "bricks." On the Altix, the computational brick consists of four Intel Itanium2 processors (in two nodes), local memory, and a two-controller ASIC called the Scalable Hub (SHUB). Each SHUB interfaces to two CPUs in one node, along with memory, I/O devices, and other SHUBs. The Altix cache-coherency protocol is also implemented in the SHUB. The 64-bit Itanium2 architecture operates at 1.5 GHz and is capable of issuing two MADDs per cycle for a peak performance of 6.0 Gflop/s. The memory hierarchy consists of 128 FP registers and three on-chip data caches (32KB L1, 256KB L2, and 6MB L3). The Itanium2 cannot store FP data in L1, making register loads and spills a potential source of bottlenecks; however, a relatively large register set helps mitigate this issue. The superscalar processor implements the Explicitly Parallel Instruction set Computing (EPIC) technology where instructions are organized into 128-bit VLIW bundles. All Altix experiments reported here were performed on the 10,240-processor Columbia system running 64-bit Linux version 2.4.21, located at NASA Ames Research Center.

3.4 Phoenix (Cray X1E)

The Cray X1E is the recently-released follow-on to the X1 vector platform. Vector processors use a dramatically different architectural approach than conventional superscalar systems to expedite uniform operations on independent data sets by exploiting regularities in the computational structure of scientific applications. The X1E computational core, called the single-streaming processor (SSP), contains two 32-stage vector pipes running at 1.13 GHz. Each SSP contains 32 vector registers holding 64 double-precision words, and operates at 4.5 Gflop/s. The SSP also contains a two-way out-of-order superscalar processor (564 MHz) with two 16KB caches (instruction and data). Four SSPs can be combined into a logical computational unit called the multi-streaming processor (MSP), and share a 2-way set associative 2MB data Ecache, a unique feature that allows high bandwidth (25–51 GB/s) for computations with temporal data locality. Note that the scalar unit operates at 1/4th the peak of SSP vector performance, but offers effectively 1/16 MSP performance if a loop can neither be multistreamed nor vectorized. This is because in a serialized segment of a multistreamed code, only one of the four SSP scalar processors within an MSP can do useful work, thus degrading the relative performance ratio to 32/1. Consequently, a high vector

[1]Peak performance shown for X1 MSP vector units. The scalar processsor in each SSP has a peak of 1.1 Gflop/s.

operation ratio is especially critical for effectively utilizing the underlying hardware.

The basic building block of the X1E is the compute module, containing four multi-chip modules (MCM), memory, routing logic, and external connectors. Two MSPs are implemented in a single MCM, for a total of eight MSPs per module that are organized as two SMP nodes. These nodes each use half the module's memory and share the network ports. The interconnect is hierarchical, with subsets of 16 SMP nodes connected via a crossbar. For up to 1024 MSPs, these subsets are connected in a 4D-hypercube topology; thereafter, the interconnect is a 2D torus. All reported X1E experiments were performed on a 768-MSP system running UNICOS/mp 3.0.23 and operated by Oak Ridge National Laboratory.

4. EXPERIMENTAL SETUP

In this section, we describe the computational problem that we solved, how we conducted a weak scaling study, and a portable infrastructure we used gather a per-process profile of computation and communication.

4.1 Problem Specification

Initially, we were interested in profiling a more complex adaptive application modeling the propagation of a nuclear flame in a Type Ia supernova [1]. The application was derived from a low Mach-number combustion code also developed by CCSE at LBNL. The supernova code required the solution of multi-level parabolic and elliptic PDEs for the reaction-diffusion part and the enforcement of the low-Mach number constraint, in addition to the hyperbolic PDEs that describe the evolution of the fluid. Also, an adaptive ordinary differential equation (ODE) solver was used to model the stiff chemistry and the code utilized a nonlinear, table-driven equation-of-state.

Our efforts to optimize the supernova code for the X1E platform mainly centered around rewriting the scalar modules so that it would vectorize better. In particular, we observed that most of the time was spent in the equation-of-state, which would vectorize but was called in a pointwise fashion. The effort to re-organize the code so that long vectors were passed rather than single data values took almost a man-month. Thereafter, we noticed that the ODE solver was taking a significant fraction of the time, but it would require substantial effort to vectorize it. The next major issue would be to optimize the multigrid solver for the elliptic and parabolic PDEs. The task looked particularly daunting when the code achieved a mere 80 Mflop/s on a 12 Gflop/s X1 MSP after the equation-of-state optimization.

At that point, we decided to profile a much simpler hyperbolic shock-tube problem since all the numerics easily vectorized. Furthermore, if we are unable to get good performance on the X1E with this simpler AMR gas dynamics application, we have little hope for problems with more complex physics. We modeled the interaction of a Mach 1.25 shock in air hitting a spherical bubble of helium, illustrated in Figure 2. This case is analogous to one of the experiments described by Haas and Sturtevant [10]. The helium is a factor of 0.139 less dense than the surrounding air which causes the shock to accelerate as it enters the bubble and subsequently generates vorticity that dramatically deforms the bubble.

We use a γ-law equation-of-state for each gas with $\gamma_a = 1.4$ for air and $\gamma_f = 1.667$ for the helium. Mixtures of the two gases are modeled using effective γ's: $\Gamma_c = \left(\frac{f}{\gamma_f} + \frac{1-f}{\gamma_a}\right)^{-1}$ for the speed of sound, and $\Gamma_e = 1 + \left(\frac{f}{\gamma_f - 1} + \frac{1-f}{\gamma_a - 1}\right)^{-1}$ for internal energy. The harmonic average used to compute Γ_c expresses the net change in volume of a mixture of the gases in terms of their individual compressibilities. The speed of sound, $c = \sqrt{\Gamma_c p/\rho}$, is used in the integration routine for defining characteristic speeds and for approximate solution of the Riemann problem to calculate fluxes. We assume that the two components of a mixed fluid cell are both at the same pressure p, which is computed from density ρ and internal energy e using Γ_e as $p = (\Gamma_e - 1)\rho e$. The formula used to compute Γ_e ensures that mixing of the two fluids at the same pressure does not change the pressure and internal energy of the composite fluid.

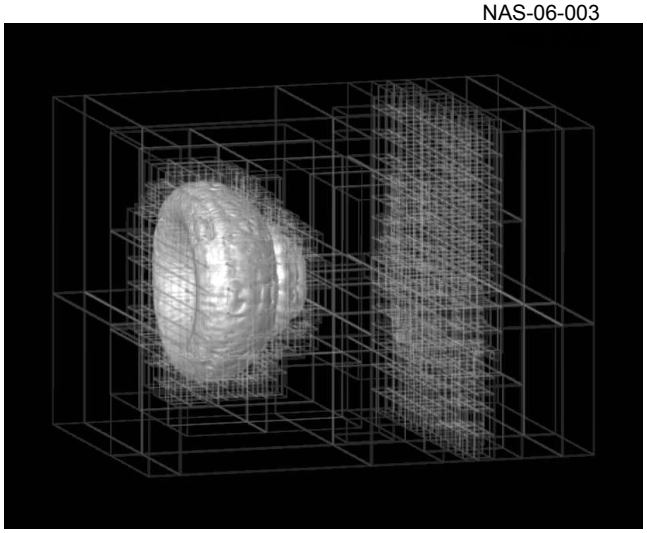


Figure 2: Volume rendering of the helium mass fraction at the end of the simulation with boxes used by the refinement scheme superimposed on the image. The grids capture the shock bubble (left) and shock front (right) as they pass out of the domain.

4.2 Weak Scaling

We decided to conduct a weak scaling study in order to best examine the parallel performance properties of the HyperCLaw AMR code when the workload per processor remained fixed. Perfect weak scaling is achieved if the number of grids and the number of refined cells at each level are proportional to the number of processors. To achieve this goal, we ran the 16-processor job a specified number of timesteps, and recorded the locations and sizes of the refined grids after each regridding phase in a log file. We used a `perl` script to read the log file and produce appropriate patched grids for each regridding step for the other runs. For example, each file for the 32-processor run contained twice the number of grids and twice the number of refined cells as the 16-processor case; however, the refined grids covered exactly the same regions of the physical domain in the two cases. This was achieved by halving the cell spacing in the X dimension for every grid, and then splitting each grid into two. The process was repeated in the Y dimension when generating grids for the 64-processor run, in the

MRL	Mem (MB) P=128	Avg Grids/P	Avg Cells/P (1000s)	Avg X-dim Grid Size	Godunov FP_CI	Overall FP_CI			
						P=32	P=64	P=128	P=256
0	438	4	1049	64	0.78	0.62	0.59	0.54	0.53
1	485	29	823	23	0.77	0.60	0.57	0.53	0.47
2	459	61	802	22	0.77	0.59	0.53	0.49	0.24

Table 2: Characteristics of AMR calculation for varying maximum refinement levels (MRL). Note that the floating-point computational intensity (FP_CI) is measured on the Seaborg system.

Z direction for the 128-processor run, and so on. In our experiments, we did not include the time required to read the log file.

We note that there are certain drawbacks to this process but emphasize that it was done primarily to conduct a weak scaling study. Halving the cell spacing in only one direction at a time generates stretched grids that are unacceptable for problems requiring multilevel elliptic solves without modifying the existing algorithms in the BoxLib/AmrLib code suite. In addition, a priori prescribing the locations of the patched grids based on a 16-processor run may not work for large production cases because the non-linearity of the method will cause important flow features to drift off the refined regions.

4.3 Integrated Performance Monitoring

Integrated Performance Monitoring (IPM) is a portable performance profiling infrastructure that binds together communication, computation, and memory information from the tasks in a parallel application into a single application-level profile [12]. IPM provides a light-weight portable mechanism for workload-wide parallel profiling that does not require user intervention and scales to 1000's of processors. As the application executes on the parallel platform, IPM records a per-process profile of computation and communication using a small fixed memory footprint and very low CPU overhead. When the application terminates, a report of the aggregate profile is generated. In this work, IPM was used on all the target architectures as a probe of the amount of communication.

The principal benefit of using IPM is that it provides sufficient contextual clarity to separately analyze the communication in each of the distinct AMR steps within HyperCLaw. Since each functional component has a specific algorithmic or data movement role in the overall calculation, having region-specific timings allows one to compare measurements with estimates derived analytically or from microbenchmarks. Analyses of parallel performance that treats the application as a whole does not provide this level of detail.

5. PERFORMANCE RESULTS

This section describes the performance results and analysis of the HyperCLaw adaptive framework modeling our shock helium bubble experiment for the three evaluated architectures.

5.1 Calculation Characteristics

An overview of the AMR calculation characteristics are presented in Table 2. Observe that for our studied problem, the average number of grids per processor increases as the maximum refinement level (MRL) grows from 0 to 2. Since each processor holds the meta-data for all grids in the problem (not just its local grids) and must search this global list to determine its required communication, the program will suffer increased overheads for managing larger numbers of grids. This will also be the case with higher concurrency experiments for a fixed MRL, since for this weak scaling configuration the aggregate number of grids increases with larger processor counts. Additionally, for our experimental setup, the average number of cells per processors decreases for an increasing MRL. This results in the advancement of fewer cells per grid, which combined with increasing grid count, increases overhead and thus reduces the calculation efficiency.

Observe that for our studied calculation the average grid's X-dimension length decreases with larger MRL. We therefore expect a decrease in vector length on the X1E associated with these small grids, as the compiler generally vectorizes across the inner-most loop of the X dimension. However, it is often the case that superscalar platforms show an increase in performance for small grids, due to increase cache reuse. Table 2 also shows the floating-point computational intensity (FP_CI) of both the Godunov solver and overall calculation as measured by Seaborg hardware counters, where FP_CI is defined as the total number of floating-point operations divided by the total loads and stores. Note that although the Godunov solver is computationally intensive, it is written in a modular fashion which focuses on program flexibility and extensibility. As a result it experiences a large number of cache misses and register spills, thus causing its surprisingly low FP_CI of approximately 0.77. The solver algorithm could in principle be restructured to improve cache behavior at the cost of reduced flexibility, but would require extensive software reengineering.

Finally, notice that FP_CI of the overall calculation is lower than just the Godunov phase. This is expected as most of the floating-point work occurs in the Godunov solver. The overall FP_CI degrades for increasing MRL and concurrencies, since the associated increase in overheads causes a smaller fraction of the runtime to be spent on the numerically intensive Godunov phase.

5.2 Architectural Performance Comparison

We now examine the performance behavior of our AMR experiment on the four architectures evaluated in our study. Table 3 presents a performance breakdown for varying MRLs and concurrency levels (the Phoenix experiment unexpectedly crashes at P=256, Cray engineers are investigating the problem). Figure 3 shows a graphical view of the main component runtimes for the MRL=2 experiment. In terms of absolute runtime, Columbia generally attains the highest performance for MRL$\geq$1 followed by Bassi, Phoenix, and Seaborg. This is somewhat surprising as the peak performance of the Phoenix vector architecture is three times that of Columbia—this discrepancy can be seen in the attained

	MRL	Seaborg (Power3)				Bassi (Power5)				Columbia (Altix)				Phoenix (X1E)		
		P=32	P=64	P=128	P=256	P=32	P=64	P=128	P=256	P=32	P=64	P=128	P=256	P=32	P=64	P=128
Total (secs)	0	856	869	893	906	127	128	133	139	129	134	159	159	51	52	58
	1	1440	1470	1513	1625	240	248	256	269	222	234	245	297	211	271	320
	2	**2207**	**2322**	**2428**	**3720**	**399**	**420**	**445**	**517**	**346**	**398**	**394**	**520**	**525**	**930**	**1433**
% of Peak	0	4.1%	4.0%	3.9%	3.8%	5.4%	5.3%	5.1%	4.9%	6.7%	6.5%	5.8%	5.5%	5.6%	5.5%	4.9%
	1	4.9%	4.8%	4.7%	4.4%	5.8%	5.6%	5.5%	5.2%	8.0%	7.6%	7.2%	5.9%	2.8%	2.2%	1.8%
	2	**5.3%**	**5.0%**	**4.8%**	**3.1%**	**5.8%**	**5.5%**	**5.2%**	**4.5%**	**8.5%**	**7.4%**	**7.4%**	**5.6%**	**1.9%**	**1.0%**	**0.7%**
Godnv (secs)	0	666	662	662	663	102	101	101	101	101	101	103	101	38	38	40
	1	1041	1034	1032	1035	172	172	172	171	142	144	141	143	96	96	96
	2	**1523**	**1521**	**1521**	**1527**	**283**	**283**	**282**	**282**	**210**	**217**	**205**	**210**	**196**	**196**	**202**
Godnv (PTT)	0	78%	76%	74%	73%	80%	79%	76%	73%	78%	75%	69%	63%	75%	74%	69%
	1	72%	70%	68%	64%	72%	69%	67%	64%	64%	62%	58%	48%	46%	35%	30%
	2	**69%**	**66%**	**63%**	**41%**	**71%**	**67%**	**63%**	**55%**	**61%**	**54%**	**52%**	**40%**	**37%**	**21%**	**14%**
Godnv % of Peak	0	4.8%	4.8%	4.8%	4.8%	6.2%	6.2%	6.2%	6.2%	7.9%	7.8%	7.7%	7.9%	6.9%	6.9%	6.6%
	1	5.7%	5.7%	5.8%	5.7%	6.8%	6.8%	6.8%	6.8%	10.4%	10.3%	10.5%	10.4%	5.1%	5.2%	5.2%
	2	**6.7%**	**6.7%**	**6.7%**	**6.7%**	**7.1%**	**7.1%**	**7.1%**	**7.1%**	**12.1%**	**11.7%**	**12.4%**	**12.1%**	**4.3%**	**4.3%**	**4.2%**
MPI (PTT)	0	5%	7%	9%	10%	7%	8%	11%	15%	7%	10%	18%	24%	13%	13%	17%
	1	6%	8%	10%	15%	7%	9%	11%	13%	8%	13%	18%	30%	23%	30%	28%
	2	**6%**	**8%**	**10%**	**35%**	**6%**	**8%**	**9%**	**11%**	**8%**	**12%**	**17%**	**28%**	**13%**	**22%**	**22%**

Table 3: Summary of AMR performance for varying concurrencies and MRL, showing the runtime, percent of theoretical peak, and percent of total time (PTT). MRL=2 results are shown in bold.

theoretical percentage of peak, where at 128 processors and a MRL=2, Phoenix achieves only 0.7% compared with 7.4% on Columbia, 5.2% on Bassi, and 4.8% on Seaborg. This extremely low fraction of peak indicates that the class of AMR computations studied in this work map poorly to the X1E vector platform. Significant code reengineering may be able to address these limitations.

We also note that the newly released Power5 Bassi system consistently achieves a higher fraction of peak than the older Power3-based Seaborg, even though the peak performance of Bassi is more than 5x that of Seaborg. This relatively impressive performance is due to improvements in the microarchitectures and component integration, including: fast memory interface relative to peak ALU speed (see Table 1), huge 36MB L3 victim cache, large number of rename registers, on chip memory controller and fabric interface, as well as improved prefetching facilities.

Observe that on all architectures, the percentage of peak decreases with increasing concurrency (for a fixed MRL) due to increased costs associated with grid management and MPI communication. However, for increasing MRLs, we see opposite performance trends on the superscalar and vector platforms. Since increasing MRLs correspond to decreasing average grid sizes (see Section 5.1), the superscalar systems benefit from increased cache reuse, while the vector platform suffers from a reduction in the vector length.

Table 3 also shows that the Godunov solver achieves a rather low fraction of peak across all of our studied architectures, achieving only 12.4%, 7.1%, 6.7%, and 4.2% on Columbia, Bassi, Phoenix, and Seaborg, respectively, for P=128, MRL=2. As discussed in Sections 2.4 and 5.1, this computationally-intensive algorithm suffers from register spilling and poor cache reuse on the superscalar systems and short vector lengths on the vector platform. Recall that since AMR calculations inherently focus on localized regions of the domain, we expect the grid size dimensions—and the corresponding vector length—to decrease with higher MRL. As a result, the solver performance on vector platforms can be severely inhibited, as in our example where the vector length is limited to only 22 when MRL=2 (see Table 2).

Since the Godunov solver is the primary phase of floating-point operations (although other regions such as TimeStep also perform some numerics), it represents the high water mark for the achievable percentage of peak. The additional AMR components necessary to maintain and regrid the hierarchical meshes, as well as the required communication, result in non-vectorizable overheads (such as pointer chasing) that decrease the codes overall efficiency. The relative effects of the AMR overheads can be seen by measuring the percent of total time (PTT) spent in the Godunov phase. As shown in Table 3, this value falls to 63% on Seaborg and Bassi, 52% on Columbia, and only 14% on Phoenix for P=128, MRL=2, thereby indicating that the incurred AMR overheads are significantly more expensive on the vector platform.

Finally, Table 3 shows that a relatively small fraction of overhead is required for communication across our studied platforms. The message-passing algorithm in this framework aggregates all message between any two sets of processors before performing the actual data transfer. This incurs the overhead of copying to (from) large MPI buffers from (to) the data structures of the individual grids, but amortizes communication latency by transmitting large messages. In terms of performance, Bassi's communication generally consumes the smallest PTT, followed by Seaborg, Columbia, and Phoenix (this is especially true of P=256, MRL=2, where all architectures other than Bassi see a precipitous increase in communication PTT overhead). The relatively high PTT of Phoenix, is a result of several factors. First, its high peak computational rate puts more pressure on the communication infrastructure. Additionally, MPI is not the most efficient communication protocol

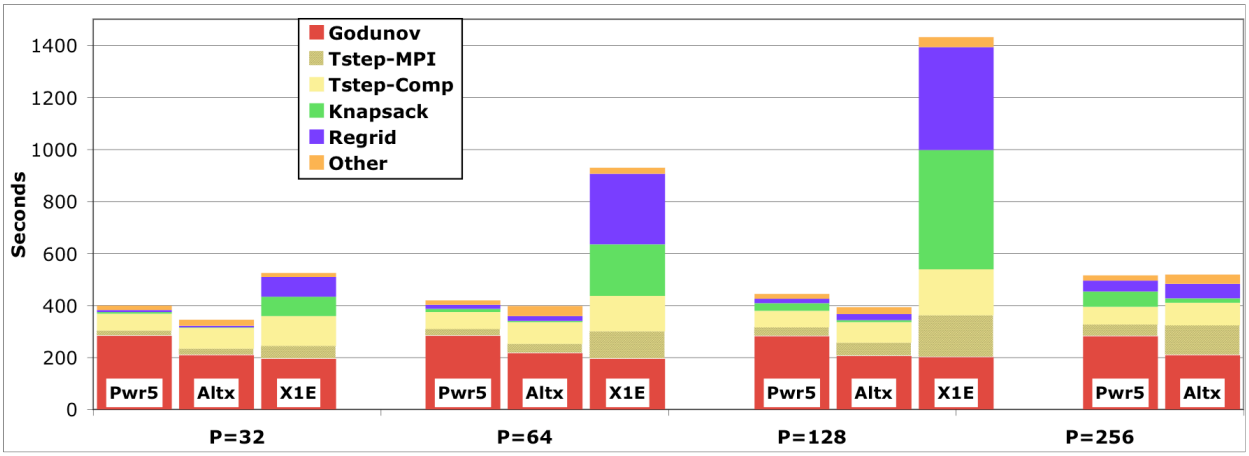


Figure 3: Breakdown of absolute MRL=2 AMR performance, showing the runtime (in seconds) contributed by the major code modules (Seaborg not shown).

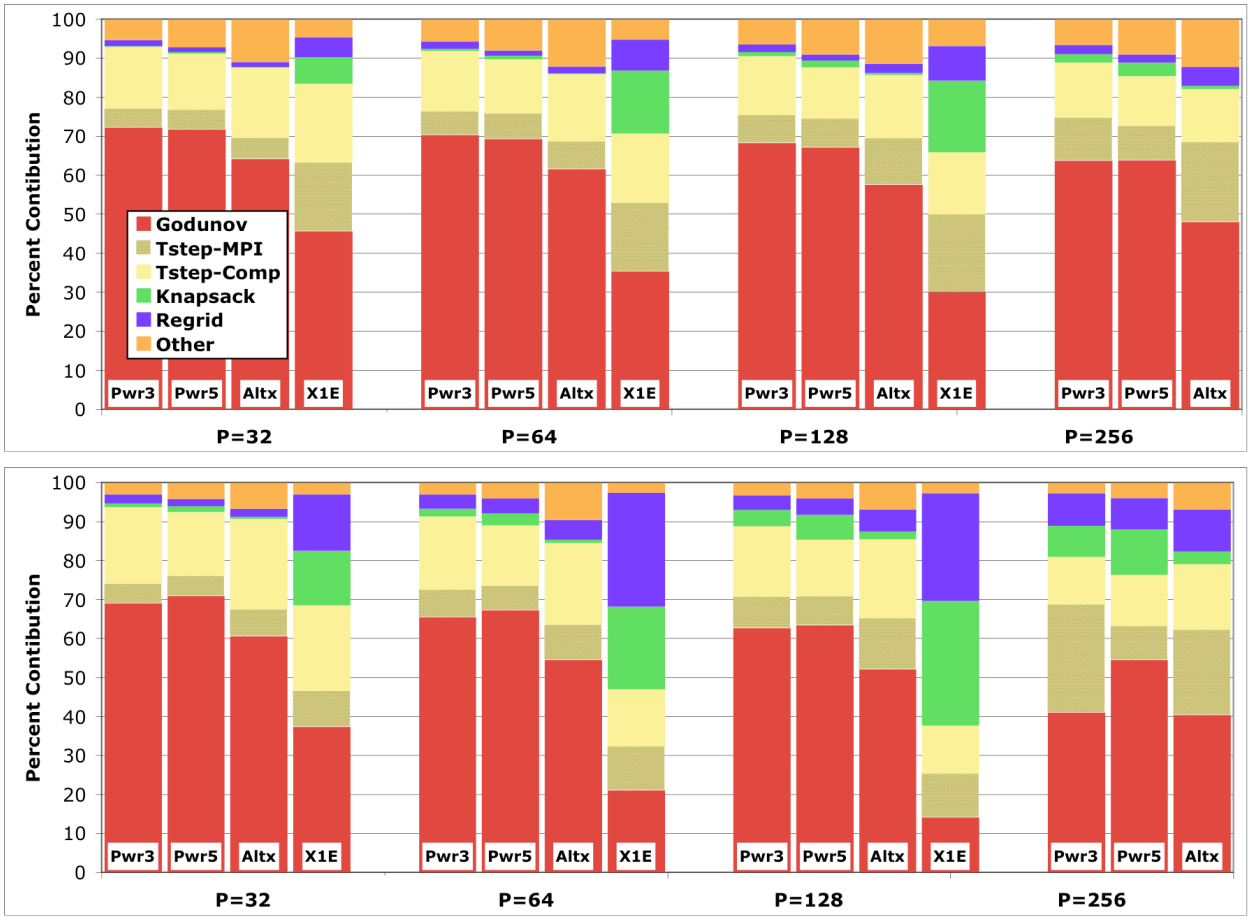


Figure 4: Breakdown of relative MRL=1 (top) and MRL=2 (bottom) AMR performance, showing the percentage of total time (PTT) contributed by the major code modules.

	Seaborg (Power3)				Bassi (Power5)				Columbia (Altix)				Phoenix (X1E)		
	P=32	P=64	P=128	P=256	P=32	P=64	P=128	P=256	P=32	P=64	P=128	P=256	P=32	P=64	P=128
Speedup	5.27	5.08	4.98	3.30	4.33	4.14	4.05	3.63	5.07	4.57	5.12	4.10	1.28	1.30	0.52

Table 4: Benefit of AMR: Speedup of AMR calculation at MRL=2 relative to non-adaptive (MRL=0) computation using a uniform grid at the finest resolution of interest.

on tightly connected, global-address space machines such as the X1E. Rewriting the communication layer using SHMEM or vector loads/stores may potentially increase performance by reducing the overhead of explicit message passing, but would inhibit code portability.

5.3 Effects of Increased Refinement Level

Figure 4 examines the PTT of the main AMR components as the maximum refinement level (MRL) increases from 1 to 2. Namely, we examine *Godunov*, *TimeStep*, *Knapsack*, and *Regrid*, where TimeStep is further subdivided into its computational and communication components (since this phase incurs most of the communication within the AMR calculation). Results clearly show that, as previously noted, increasing the MRL and/or concurrency causes a smaller portion of the computation to be spent in the numerically intensive Godunov solver. This is particularly true for the Phoenix and Seaborg systems at P=256.

Examining the individual AMR components, we see that that the overhead associated with the TimeStep computational phase shows almost no increase with a growing MRL. This is also true of Timestep's communication component. Since the studied code is not communication intensive, other components of the computation consume relatively larger fractions of the runtime as the maximum refinement increases.

The Knapsack overhead is due to its asymptotic $O(N^\kappa)$ cost with $2 < \kappa < 3$, where N is the total number of grids in the computation. Observe, however, that on the superscalar systems this cost is relatively minor even at the highest concurrency and MRL of two—the Columbia system in particular achieves extremely impressive performance for this integer-intensive calculation, consuming less than 2% of the overhead. The Knapsack algorithm heavily relies on the C++ Standard Template Library for list insertions, deletions, traversals, etc. Clearly, the Columbia system has an advantage in compiling and running this class of codes. However, the Phoenix platform suffers greatly with increasing processor count and MRL, due to the large fraction of non-vectorizable code portions of this integer- and pointer-based algorithm. For example at P=128, the overhead of the Knapsack routine grows from 18% to 32% of the overall runtime.

Similarly, the Regrid phase accounts for a modest portion of the superscalar overhead (less than 6% PTT for P=128, MRL=2), but is responsible for a significant fraction of overhead on Phoenix. The Regrid algorithm is not well suited for vector systems as it consists of numerous list traversals, non-vectorizing integer operations, and short vector length interpolations. This is particularly true as more, smaller grids are generated with increasing MRL. Thus on the vector platform, we see the Regrid PTT increases from 9% to 28% of the overhead for 128 processors for an MRL=2. Therefore, the combined Knapsack and Regrid phases are responsible for over 60% of the overhead on Phoenix (for P=128), compared with less than 8% on the superscalar platforms. Extensive code reengineering may address these deficiencies, but would require an AMR framework developed specifically from the ground up with vector optimizations in mind. Furthermore, if additional levels of refinement continue to increase the number of grids, while reducing the average grid size (as in these experiments), we would expect these overhead trends to continue with increasing MRL.

6. CONCLUSIONS

In this work, we described a detailed performance analysis of the HyperCLaw AMR framework on leading scalar and vector platforms: the IBM Power3 and Power5, SGI Altix, and Cray X1E. Table 4 presents a performance comparison of the target architectures for our AMR calculation based one of its most important metrics: the time-to-solution speedup of using the AMR framework versus the time it would have taken to perform the calculation using a uniform (non-adaptive) mesh at the finest resolution of interest. Observe that on the superscalar platforms, a speedup of 5.1x, 5.0x, and 4.1x are achieved on Columbia, Seaborg, and Bassi respectively (at P=128) for our weak-scaling MRL=2 experiment. The X1E vector system, on the other hand, attains a slight AMR advantage of 1.3x for small numbers of processors, and losses the AMR computational benefit all together for P=128 as the speedup ratio falls to only 0.5.

This very low performance indicates that the studied AMR implementation maps poorly to the Phoenix vector system even though much of the code was originally developed on the Cray YMP/C90 architecture of the early 1990s[2]. Since that time, the infrastructure has grown in sophistication and complexity in order to accommodate algorithms for low Mach number and chemically reacting flows. A reengineering of the algorithms to take advantage of the vector hardware could improve performance in some areas, but at the expense of readability and performance portability. Such an activity would be a substantial undertaking given that the underlying infrastructure of this code base now contains over 100,000 lines of C++. Finally, it is unclear how much of a performance improvement would ultimately be possible, given that some scalar computations would be required even in a vector-optimized code implementation.

In general, it is difficult to make statements on the performance of an AMR application primarily because there is such a wide diversity of cases that can arise. For example, it is not always true that one can extrapolate the performance characteristics of a MRL calculation to a MRL + 1 calculation; such predictions would depend on the physical problem being studied, among other factors. A turbulent flame calculation might well increase in the number of grids and work done with additional AMR levels being added to the calculation, but some astrophysical applications actu-

[2]The Cray C90/YMP shared-memory systems were more balanced in their scalar and vector performance compared to the X1E and multiprocessor performance was obtained using Cray threads rather than message passing [7].

ally have fewer grids as the number of AMR levels increase. However, for the specific problem examined in this paper, we would indeed expect the overheads costs to continue growing with increasing MRL, thus causing the Godunov solver to account for diminishing portions of the overall runtime.

Furthermore, recall that our original goal was to study a much more complex adaptive algorithm that examined the propagation of a nuclear flame in a Type Ia supernova. This supernova calculation requires the use of multi-grid solvers to advance the solution of multi-level parabolic and elliptic PDEs. Unfortunately, these advanced physics and chemistry calculations would further reduce the vector length, since they require the solution of progressively coarser grids, as well as interpolation within and between the multi-grid hierarchies.

Essentially, changes would be required in two types of software modules: the multigrid iterative solvers, and with the method used to integrate the ordinary differential equations (ODEs) used in the combustion of nuclear fuel. The performance of the AMR multigrid solvers on vector machines is hampered by the increasingly shorter vector lengths that occur in the prolongation, restriction, and smoothing phases. To some extent, this problem can be addressed by arranging the sub-grids on a processor into a pseudo-subgrid and applying the operation over the larger grid via a vector mask. Replacing the ODE solvers, which operate in a pointwise fashion, with a version that operates on vectors of points is also possible, but due to the variability in the convergence properties of the ODE solver with nuclear fuel composition, high efficiency might be difficult to attain. We thus conclude that it would be extremely challenging to achieve high vector performance on our full-scale AMR application of interest.

The studies described in this work are being used to guide new research activities in improving the performance of AMR applications on modern HEC systems. One path aims to lower the computational cost and storage requirement of the meta-data; primarily in the message-passing overhead and in the Knapsack load balance scheme. This will also improve the performance of the supernova and low-Mach number combustion codes. Another research direction will attempt to reorganize the Godunov solver to improve memory-access patterns and cache reuse. As the issues relating to performance of HyperCLaw are resolved, these more sophisticated applications will be the target of our research in understanding and optimizing performance characteristics of AMR behavior on HEC systems.

Acknowledgments

The authors sincerely thank ORNL for providing access to the X1E. All authors from LBNL were supported by OASCR in the DOE Office of Science under contract DE-AC03-76SF-00098.

7. REFERENCES

[1] J. B. Bell, M. S. Day, C. A. Rendleman, S. E. Woosley, and M. A. Zingale. Adaptive low mach number simulations of nuclear flame microphysics. *Journal of Computational Physics*, 195(2):677–694, 2004.

[2] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, 1989.

[3] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[4] Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory. `http://seesar.lbl.gov/CCSE`.

[5] Chombo software package for AMR applications design document, September 2003. `http://seesar.lbl.gov/anag/chombo/ChomboDesign-1.4.pdf`.

[6] P. Colella. A direct Eulerian MUSCL scheme for gas dynamics. *SIAM Journal on Scientific and Statistical Computing*, 6:104–117, 1985.

[7] P. Colella and W. Y. Crutchfield. A parallel adaptive mesh refinement algorithm on the C-90. In *Proceedings of the Energy Research Power Users Symposium*, July 1994.

[8] W. Y. Crutchfield. Load balancing irregular algorithms. Technical Report UCRL-JC-107679, Lawrence Livermore National Laboratory, July 1991.

[9] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. A. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131:273–334, 2000.

[10] J.-F. Haas and B. Sturtevant. Interaction of weak shock waves with cylindrical and spherical gas inhomogeneities. *Journal of Fluid Mechanics*, 181:41–76, 1987.

[11] HPC Challenge benchmark. `http://icl.cs.utk.edu/hpcc/index.html`.

[12] IPM Homepage. `http://www.nersc.gov/projects/ipm`.

[13] ORNL Cray X1 evaluation. `http://www.csm.ornl.gov/~dunigan/cray`.

[14] M. Parashar and J. C. Browne. `http://www.caip.rutgers.edu/~parashar/DAGH`.

[15] C. A. Rendleman, V. E. Beckner, M. J. Lijewski, W. Y. Crutchfield, and J. B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms. *Computing and Visualization in Science*, 3(3):147–157, 2000.

[16] SAMRAI home page. `http://www.llnl.gov/CASC/SAMRAI`.

[17] STREAM home page. `http://www.cs.virginia.edu/stream`.