Scalability Challenges for Massively Parallel AMR Applications

Brian Van Straalen*, John Shalf†, Terry Ligocki*, Noel Keen*†, Woo-Sun Yang†
* *ANAG, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA*
† *NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA*

Abstract

PDE solvers using Adaptive Mesh Refinement on block structured grids are some of the most challenging applications to adapt to massively parallel computing environments. We describe optimizations to the Chombo AMR framework that enable it to scale efficiently to thousands of processors on the Cray XT4. The optimization process also uncovered OS-related performance variations that were not explained by conventional OS interference benchmarks. Ultimately the variability was traced back to complex interactions between the application, system software, and the memory hierarchy. Once identified, software modifications to control the variability improved performance by 20% and decreased the variation in computation time across processors by a factor of 3. These newly identified sources of variation will impact many applications and suggest new benchmarks for OS-services be developed.

1 Introduction

As processor clock rates have stalled, future performance improvements for scientific applications increasingly depends on scaling HPC systems to unprecedented numbers of processors. The move from exponentially improving clock rates towards exponential increases in system parallelism puts stress on every aspect of HPC system design and raises fundamental questions about languages and application programming models. Since future application performance scaling is increasingly dependent on massive parallelism, it is critically important that both applications and algorithms are reformulated to utilize these dramatic increases in system concurrency. Applications and algorithmic approaches that fail to have good parallel scaling efficiency will be increasingly marginalized as system parallelism will double every 18 months for the foreseeable future.

PDE solvers using adaptive mesh refinement, AMR, on block structured grids, e.g. [1,2], are among the most challenging applications to adapt to massively parallel computing environments. AMR is typically discounted as being inherently unscalable due to complex load balancing demands and intense communication requirements. This paper counters such arguments by describing optimizations to the *Chombo AMR framework* [4] that enable it to scale efficiently to thousands of processors on the Cray XT4. A critical component of our study was developing a highly instrumented implementation of the Chombo code and appropriate test problem configurations that provide a basis for understanding the advantages and disadvantages of our approach.

The instrumented version of Chombo, which was used to analyze scaling bottlenecks, uncovered sources of performance variation that were directly attributable to changes in the OS rather than any aspect of the hardware or application code. In particular, when one Cray XT4 system used for testing was upgraded from the Catamount micro-kernel to Compute Node Linux (CNL), we observed a 10% drop in application performance and an increase of the coefficient of variation (CoV) of individual MPI task runtimes by a factor of 7, which was directly responsible for the drop in application performance. However, the same sources of variation were not evident when measured by traditional OS variation benchmarks, such as P-SNAP [5], which cast doubt upon an OS-interference hypothesis. Ultimately, we found the source of variability was complex interactions between the Linux/GNU libc heap management algorithm and the memory hierarchy that were extremely difficult to detect using conventional means. Once identified, software modifications to control the source of variability improved performance by 20% and decreased the CoV of individual MPI task runtimes by a factor of 3.

In this study, we demonstrate that with appropriate code instrumentation and code restructuring, AMR codes can indeed be optimized to achieve scalable performance on leading HPC systems. In so doing, we have demonstrated scalable performance on up to 8192 processors on the Cray XT4, which is the best performance to date for an AMR hyperbolic gas dynamics solver, one

of the most demanding of our benchmark problems [3]. We also show that with increasing system concurrency, sources of system variation have an increasingly substantial impact on the performance of a broad range of bulk-synchronous parallel applications. A unique contribution of this paper is the analysis of a novel source of system variability that is related to the OS services such as memory allocation management rather than the kernel itself. The difficulty of diagnosing this problem and the ineffectiveness of existing OS interference benchmarks for diagnosing this problem suggests the need to develop new benchmarks that capture this new class of variability in OS services.

2 Experimental Testbed

These experiments were run on two Cray supercomputers over the period of 8 months: *Franklin* at LBNL and *Jaguar* at ORNL. *Franklin* is a 9,660 node XT4 system operated by the National Energy Research Scientific Computing Center (NERSC) with two cores per node. *Jaguar* is as a hybrid Cray XT system consisting of 5,294 dual-core XT4 nodes and 6,206 dual-core XT3 nodes, which is operated by the ORNL National Center for Computational Sciences (NCCS).

2.1 Hardware: Cray XT4 and XT3

Each node of the Cray XT4 contains a dual-core 2.6 GHz AMD Opteron processor for a total of 19,320 compute processors on the entire *Franklin* system. The memory subsystem uses DDR2-667MHz memory, which offers nearly 7GB/s aggregate memory bandwidth per core when measured by the STREAM Triad memory bandwidth benchmark. The processors are tightly integrated to the XT interconnect via a Cray SeaStar 2.1 ASIC through a 6.4GB/s bidirectional HyperTransport interface. All the SeaStar routing chips are interconnected in a 3D torus topology, where each node has a direct link to six of its nearest neighbors on the torus with a peak bidirectional bandwidth of 7.6GB/s. Both the processors and node architecture of *Jaguar* and *Franklin* are nearly identical, giving us an opportunity to compare the CNL operating system to the Catamount micro-kernel.

Some of our experiments were conducted on XT3 nodes on the NCCS *Jaguar* system, which use the same 2.6 GHz AMD Opteron processor cores and interconnect topology. However, the older-model XT3 nodes use slower DDR1-266MHz memory, which offers about half the effective memory bandwidth of the XT4's DDR2 memory subsystem.

2.2 Operating Systems

Cray XT series systems have been offered with two different operating systems; Catamount, which is a specialized micro-kernel OS, and Compute Node Linux (CNL), which is based on a Linux kernel. During our experiments, both systems initially ran Catamount and subsequently migrated to CNL. This transition provided us with a unique opportunity to directly assess the effect of the OS kernel on system variability and delivered system performance for both the Catamount and CNL operating environments.

The Catamount micro-kernel was developed by Sandia National Laboratory for the Red Storm computing system [6]. The simpler kernel design of Catamount minimizes the memory footprint of the kernel and controls for a wide range of sources of OS-induced variation. One deficiency of Catamount is that it does not support symmetric access to the device interface, or support shared memory for symmetric multiprocessing. Another deficiency is that one of the two processors on the socket must handle all I/O operations on behalf of both processors, leading to a slight load imbalance. The Catamount kernel does not currently support the quad-core barcelona processor, which motivated NERSC and ORNL to migrate to CNL.

CNL is a lightweight kernel based on the Linux OS. The CNL design also pays special attention to making many sources of OS interference quiescent, such as cron jobs and other stochastically scheduled daemon processes that exist in a full Linux OS implementation. CNL provides many of the familiar OS services present in a desktop Linux implementation, such as support for symmetric multiprocessing, shared memory, and most conventional system libraries. However, the compute nodes of the XT4 offer a very restricted environment that does not include dynamic linking, advanced scripting languages, or shell environment features. The shared memory support will enable use of hybrid programming models using OpenMP, and allow each of the cores symmetric access to the communication interface.

3 Application

Block-structured AMR, developed by Berger and Oliger [1, 2] for computational gas dynamics, is a multiscale algorithm that achieves high spatial and temporal resolution in localized regions of dynamic multidimensional numerical simulations. A broad range of physical phenomena modeled by PDE exhibit multiscale behavior where variations in the solution occur over scales that are much smaller than the overall problem domain. Examples include flame fronts arising in the burning of

hydrocarbon fuels, nuclear burning in supernovae, effects of localized features in orography or bathymetry on ocean currents, tracking of tropical cyclones, localized kinetic effects for plasma physics problems, and, in general, small scale effects due to nonlinear instabilities. In each of these problems, the fundamental mathematical description is given in terms of various combinations of PDE of classical type (elliptic, parabolic, hyperbolic).

The Berger and Oliger AMR algorithm organizes refined regions into rectangular structured grids of several hundred to several thousand grid points per grid. High-resolution structured-grid methods (typically expressed as stencils) are used to advance the solution in time. Furthermore, the overhead of managing the irregular data is amortized over a relatively large number of floating point operations on the rectangular grids. For time-dependent problems, refinement is performed in time as well as space. Each level of spatial refinement has its own stable time step, with the time steps on a level constrained to be integer multiples of the time steps on all finer levels.

3.1 Chombo AMR Framework

AMR applications require a long-term sustained investment in software infrastructure to create scalable solvers that are capable of utilizing the full capabilities of the largest available HPC platforms. We have created a framework for implementing scalable parallel AMR calculations called *Chombo* [4] that provides an environment for rapidly assembling portable, high-performance AMR applications for a broad variety of scientific disciplines.

Chombo is a fully instrumented C++ library. There are a set of timer macros that can be used to time functions or sections of code. These timers attempt to use native instructions on the target architecture in order to minimize the overhead of collecting detailed performance data. In the case of the Cray XT, Chombo measures elapsed time using the `rdtsc` x86 assembly instruction. This returns a 64-bit unsigned integer representing the number of clock cycles since last processor reset. On the AMD Opteron processor this instruction takes, on average, 13 cycles to execute, and for a 2.6GHz clock provides a 0.385 nanosecond timing resolution. The timers collect information which is summarized and output at the end of a run for each processor. The output of the Chombo timing infrastructure is similar in format to the GNU profiler (`gprof`) output but only instrumented functions and sections of code are reported. This information is then post-processed to produce various statistics, e.g., mean/min/max times, standard deviations, and correlations.

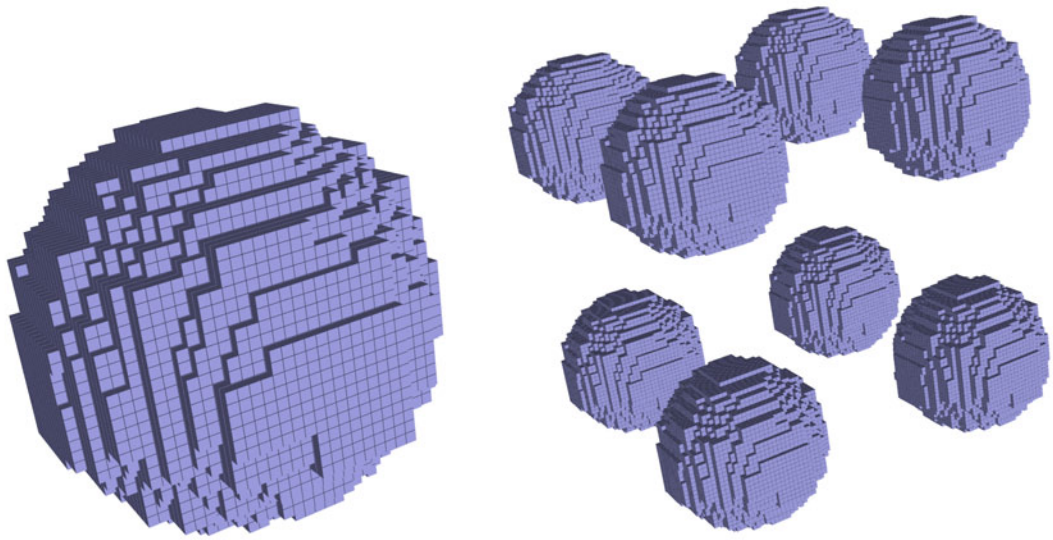
Additional analysis was performed using the `CrayPat` performance analysis tools on the Cray XT system. Specifically, certain sections of code were manually instrumented with `CrayPat` calls.

4 Benchmarking Methodology

In many applications that use PDE solvers, the primary motivation for using large numbers of processors is to achieve weak scaling. Even with AMR, many leading scientific problems remain out of reach due to inadequate grid resolution. In those cases, increasing the number of processors is used to increase the spatial resolution of the grids using the minimum number of processors necessary to fit the problem into the available memory. Therefore, we focus on a methodology for constructing weak-scaled AMR benchmarks because this methodology models the dominant use-case for scientific problems that employ this computational method.

4.1 Replication Scaling Benchmarks

Classically, weak scaling studies of numerical methods for solving PDE on uniform grids have been performed using mesh refinement, which involves scaling the problem, refining the grid by an integer factor in each direction and increasing the number of processors so that the number of grid points per processor is fixed. The analogous scaling method for AMR would refine the coarsest grid by an integer factor and decrease the error tolerance so that the resolution at each level is increased by the same integer factor. In practice, such an approach leads to scaling behavior that is difficult to interpret. Under such a refinement scheme, the size of the refined regions at each level can change significantly, which often decreases the physical size of the refined region at a given level. Therefore, the data-dependent behavior of the AMR refinement heuristics can cause changes in AMR scaling performance that are difficult to distinguish from loss in scaling due to other causes. For this reason, we have developed benchmarking methods based on *replication scaling* which take a grid hierarchy and data for a fixed number of processors and scale it to higher concurrencies by making identical copies of the hierarchy and the data, see Figure 1. The full AMR code (processor assignment, problem setup, etc.) is run without any modifications to guarantee it is not directly aware of the replicated grid structure. Replication scaling tests most aspects of weak scalability, is simple to define, and provides results that are easy to interpret. Thus, it is a very useful tool for understanding and correcting impediments to efficient scaling in an AMR context. Furthermore, it is a good proxy for the scaling behavior of real applications. For example, a large part of



(a) (b)

Figure 1. (a) Grids at the finest AMR level used in the hyperbolic gas dynamics benchmark – these grids cover the shock front of a spherical explosion in 3D. (b) Replicated grids at the finest AMR level used in the weak scaling performance study of the hyperbolic gas dynamics benchmark.

the simulation of a gas turbine will be the simulation of multiple identical burners arranged in a ring.

Replication scaling does not rigorously test load balancing. Load imbalances that are inherent at smaller scales tend to remain the same as regions are replicated to scale the problem. Thus, the results obtained using replication scaling need to be supplemented with other measurements to obtain definitive scaling behavior, such as showing that the $t_{wall} \times N_{proc}$ divided by the number of grid points ("grind time") is bounded in the weak scaling limit using a more traditional AMR mesh refinement study [10, 13]. This work follows very closely the approach taken in [12].

4.2 Hyperbolic Gas Dynamics Benchmark

We benchmarked an explicit method for unsteady inviscid gas dynamics in three dimensions that is based on an unsplit PPM algorithm [7, 14]. This algorithm requires approximately 6000 flops/grid point. Since it is an explicit method, communication between processors is required only once per time step. We used the implementation of this method from the Chombo software distribution without significant modification. The operator peak performance for this method on the Cray XT4 was 530 Mflops/processor. The initial grids used for the replication benchmark came from a 3D, spherical-shock problem with the finest grids covering the spherical shock, see Figure 1.

The benchmark used three levels of AMR with a factor of 4 refinement between levels and with refinement in time proportional to refinement in space. We use fixed-sized 16^3 grids and a total of 6.2×10^7 grid points and five unknowns per grid point, with 10^9 grid point updates performed for the single coarse-level time step. None of the grids at any level were changed during any of the time steps, i.e., there was no grid adaptation in time which is sometimes called "regridding". In the results given here, we are only timing the cost of computing a single coarse-level time step, which includes all intermediate and fine time steps on all AMR levels but excludes the problem setup and initialization times.

5 Optimizing AMR for Scalability

Load balancing and communication volume are often blamed as the leading impediments to AMR performance scalability, but careful profiling and analysis of the code performance showed such concerns to be the lowest priority relative to other scalability bottlenecks that were identified. Indeed, many of the scaling problems related to early design decisions for the grid management infrastructure that had an inconsequential performance impact at low concurrencies became major bottlenecks as the code was scaled to thousands of processors. We begin by describing the code optimization strategies and then discuss new sources of OS/system variability and its impact on AMR performance.

5.1 Baseline Code Optimizations

The code optimizations that improved our AMR scaling behavior fall into three major categories: improving communication locality, converting to metadata management algorithms with O(N) computational complexity, and optimizing coarse-fine boundary value computations.

5.1.1 Minimizing Communications Costs

We found it necessary to distribute patches in a way that minimizes communications costs using space-filling curves. If D is the spatial dimension of the problem, Morton ordering [8] is a 1-1 mapping of $\mathbb{Z}^D$ onto $\mathbb{Z}$ with good locality, i.e., the fraction of nearest neighbors in $\mathbb{Z}^D$ of the inverse image of an interval $\mathcal{I} \subset \mathbb{Z}$ of length M whose Morton indices are not in $\mathcal{I}$ is $O(M^{-1/D})$. Load balancing is done by sorting the patches according to the Morton indices of their low corners, and dividing the linearly-ordered patches into intervals with equal workloads.

Morton ordering was chosen because it can be computed very efficiently and guarantees good locality within a single level of refinement, i.e., intra-level. The partitioning onto processors obtained using Morton ordering shows uniform distributions with only a small fraction of the patches having neighbors that are off-processor. This is in contrast to a recursive bisection approach that was used previously, in which it is possible to have long thin partitions with all the patches requiring boundary data from off-processor.

In practice, Morton ordering also produced good locality between levels of refinement, i.e., inter-level, for our AMR computations. Also, Morton ordering makes it advantageous, especially at high concurrencies (4096+ processors), to overlap the local copying of data from neighbors that are on the same processor with the remote copying of data from neighbors that are on other processors using asynchronous MPI calls. Other self-similar, space-filling curves should have similar advantages.

5.1.2 Scalable Computation of Patch Metadata

There were a number of bottlenecks that were related to data management algorithms of O(N^2) complexity that were insignificant at small concurrencies, but rapidly grew into major scaling bottlenecks for thousands of processors. Our study uncovered numerous examples where expedient programming choices early in the development of the framework revealed themselves at higher concurrencies. One such example is the management of patch metadata in the complex and dynamically changing grid hierarchies.

In our current implementations of AMR, every processor has a copy of the metadata which consists of all the patch outlines and processor assignments. These are used to compute intersection lists, e.g. which patches/processors contain neighbor data, which must be periodically copied. When using thousands of processors, it is essential to use $O(log(N_{patch}))$ sorts and searches to compute these intersection lists. Otherwise, there is a catastrophic failure to scale due to performing $O(N_{patch})$ computations on every processor. Even using fast methods, the cost of these computations are not negligible, so significant performance improvements were obtained by caching intersection lists.

5.1.3 Optimizing Coarse-Fine Boundary Condition Calculations

Coarse-fine boundary conditions involve parallel communication and irregular computation. While these calculations are scalable, they can substantially impact the variability of code runtimes and are difficult to account for accurately in the load balancing. The approach we used was to highly optimize the irregular computations and thus minimize their overall impact. We make aggressive use of residual-correction forms of the PDE to minimize how often the coarse-fine boundary conditions are computed. For interpolation stencils that are regular, we call Fortran implementations of these stencils. Although we have not done so here, we could also have taken advantage of fixed-size patches to make nearly all such calculations regular or develop fast irregular stencil operations.

5.1.4 Load Imbalances

The hyperbolic gas dynamics benchmark highlighted several anomalies in the timing results that initially appeared to be load imbalances in the core computational kernel. This core computational kernel was a discretization of a hyperbolic PDE, which can be broken down into three broadly-defined phases of computation; *pre-advance*, *time advance*, and *post-advance*. The *time advance* phase is of primary interest because it contains no communication or I/O but does perform most of the computation. This phase executes once at the coarsest AMR level, four times at the intermediate level, and sixteen times at the finest level. It was straightforward to monitor sources of variability by inserting barriers between individual phases of computation. Timing the entry and exit to these barriers enabled direct measurement of any load imbalances and the isolation of sections of code that contributed to the imbalance.

The resulting measurements showed that nearly all the significant load imbalances were in the *time advance*

phase, which was unexpected because all the grids on all levels were the same size, 16^3 elements, each processor had almost the same number of grids, and the work done on each grid should have been insensitive to the data. For example, at the finest level, each processor had 232 or 233 grids, which should have resulted in a maximum 0.4% imbalance. Separately conducted sensitivity experiments found that there could be a maximum 3.5% worst-case variation in runtime per grid depending on the data present, but this could only occur under conditions that were unlikely to be found in actual computations and did not occur during our benchmarks.

Further comparisons between the *Jaguar* system running Catamount (*Jaguar*/Catamount) and the *Franklin* system running CNL (*Franklin*/CNL) led us to believe OS interference was at fault. Specifically, on *Jaguar*/Catamount the maximum runtime on a few processors for the *time advance* phase was much greater than all the other processors, well above the mean by several standard deviations. Moreover, this behavior occurred much more frequently on large runs. On *Franklin*/CNL, and later on *Jaguar*/CNL, the mean runtime was consistently higher, by about 10%, and the CoV was 7 times greater than typical runtimes on *Jaguar*/Catamount. Uncovering the source of variation became the subject of a much deeper analysis of the AMR code's interaction with the software and OS environment on the Cray XT4 systems.

5.2 Sources of Runtime Variability

In our attempts to correct load imbalances that were discovered in the Chombo code, we uncovered sources of variability that were originally mis-attributed to algorithm characteristics. This section traces our approach to isolating sources of runtime variability on the *Franklin* and *Jaguar* systems.

5.2.1 Jaguar Runtime Variability

Runtime variability caused by stochastic sources of interference can substantially impact the runtime of bulk synchronous parallel applications such as Chombo. Figure 2(a) shows comparative runtimes for each MPI processor rank for identical code running on *Jaguar*/Catamount XT3/XT4 and *Franklin*/CNL. Although each processor was given a nearly identical workload, the *Franklin*/CNL nodes exhibited considerable variability in runtime performance as shown by the very broad (noisy) red line. The *Jaguar*/Catamount performance for both XT3 and XT4 systems, shown by the blue and green lines, is much less noisy, but shows small spikes in the graph where individual MPI ranks were delayed substantially compared to their peers. Given the bulk-synchronous nature of this code, the worst-case delays end up determining the overall runtime on *Jaguar*/Catamount. So, although the average performance of the *Jaguar*/Catamount XT4 system was substantially better than *Franklin*/CNL, the delivered performance was no better.

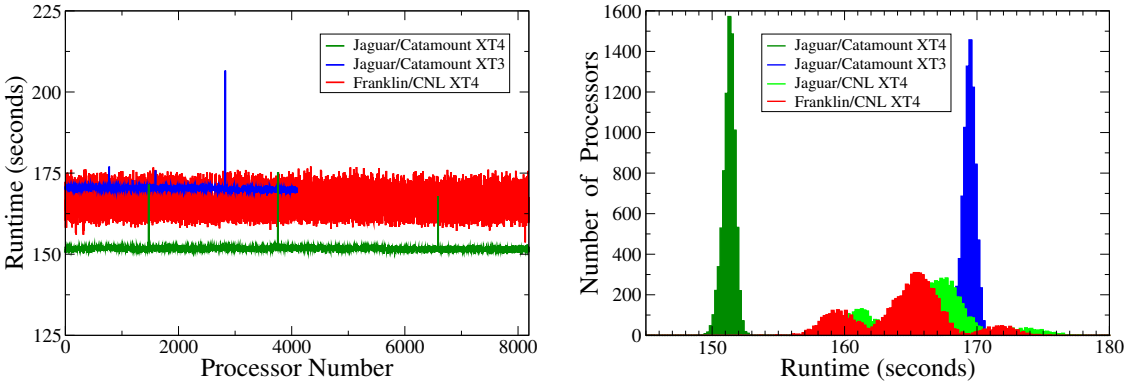
A closer inspection of the data showed that the spikes were always two processors on the same node, leading to an initial theory that XT3 nodes were allocated by mistake. However, the XT3 runs also showed anomalous spikes in node performance, shown by the blue spikes in Figure 2(a), that were proportionally higher than those of the nominal XT3 nodes. An examination of the batch logs showed that no XT3 nodes participated in the XT4 jobs. The ultimate cause of the problem was tracked down to the overhead of the ECC correction of single-bit errors in the memories of the affected nodes, leading to measurable load imbalances. Both ORNL and NERSC were able to correct the problem by increasing the voltage of the memory subsystems to the point that the memory errors were all but eliminated. Subsequently, the spikes observed in Figure 2(a) no longer occurred.

5.2.2 Franklin Runtime Variability

The primary source of variability was corrected on *Jaguar*/Catamount XT3/XT4, but persisted on *Franklin*/CNL. *Franklin*/CNL was now 8.7% slower than the *Jaguar*/Catamount XT4 and had a CoV that was 7 times larger. Figure 2(b) shows node completion times plotted as a histogram. Unlike Figure 2(a), this representation shows a clear structure in the variation on *Franklin*/CNL as three distinct Gaussian distributions, i.e., a tri-normal distribution. Given the *Franklin*/CNL and *Jaguar*/Catamount systems were nearly identical in architecture, any differences in behavior were most likely due to differences in the software environment.

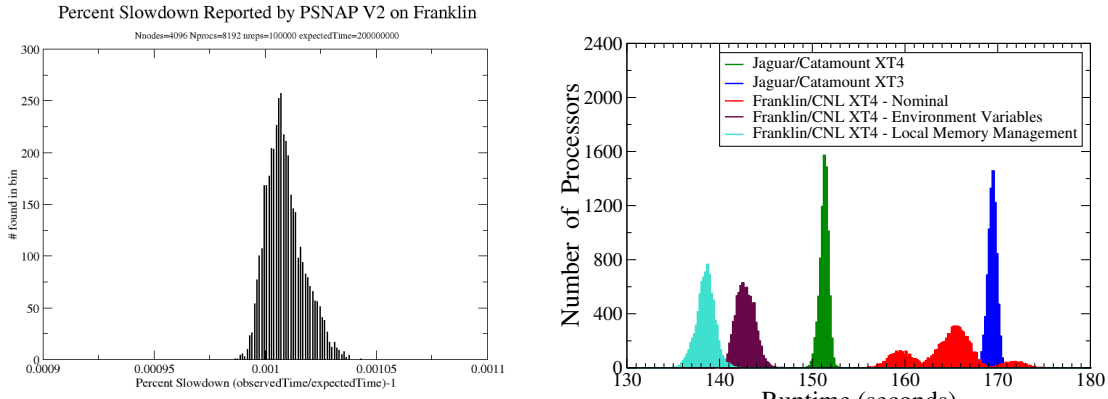
One early hypothesis was that we had uncovered a classic case of OS interference that was analyzed in detail on the ASCI White and ASCI Q systems [9]. Therefore, we obtained an operating system benchmark developed at LANL named P-SNAP, which uses one method of quantifying interference or noise in operating systems. P-SNAP uses a *Fixed Work Quantity* (FWQ) approach consisting of a simple loop of constant integer additions. All processors perform the exact same operations and measure the time (by default, using `MPI_Wtime()`). On a system with no interference, each processor would report the same measured time.

P-SNAP was run on 8192 processors of *Franklin*/CNL using the default input parameters. The results shown in Figure 3(a) demonstrate modest OS noise. However, the variation in runtime measured



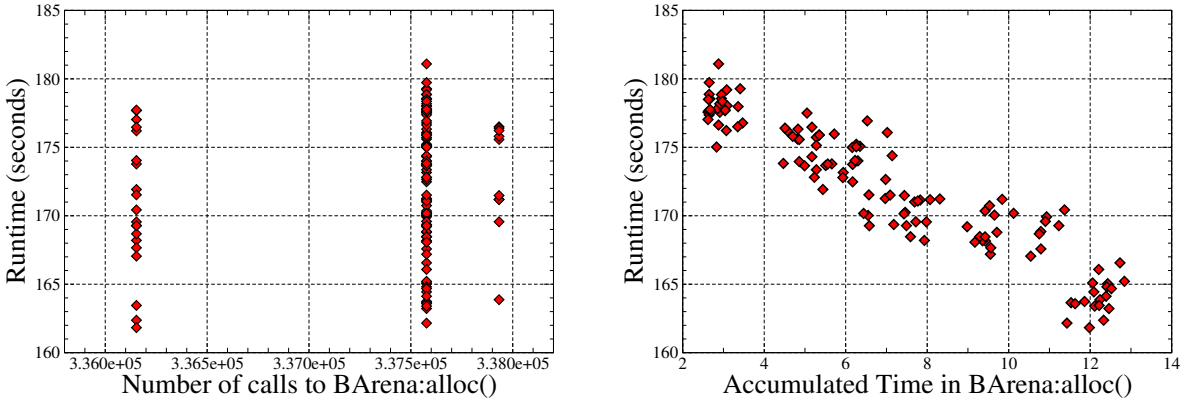
(a) (b)

Figure 2. (a) Runtime of each MPI process rank on *Jaguar*/Catamount and *Franklin*/CNL. The spikes in the *Jaguar*/Catamount runtimes were attributed to ECC memory correction overhead on nodes experiencing high memory error rates. (b) Histogram of runtime variability for the AMR hyperbolic gas dynamics runs on the evaluated systems.



(a) (b)

Figure 3. (a) 8192-way P-SNAP results on *Franklin*/CNL. (b) Histogram of AMR hyperbolic gas dynamics runtime with memory optimizations.



(a) (b)

Figure 4. Scatterplot (a) shows no correlation between the run time variation and the number of memory allocations performed. Scatterplot (b) shows a clear anti-correlation between the time spent in the memory allocation routines and the runtime variation – meaning that processors that showed the slowest runtime spent the least time in performing memory allocation, despite a balanced workload.

by P-SNAP is three orders of magnitude smaller than the variation measured by our AMR benchmark, and does not exhibit the distinctive tri-normal distribution observed in Figure 2(b). Therefore, classic OS interference is unlikely to be the cause of the observed variation in application performance. At least, if it was OS interference, a clean and straightforward benchmark like P-SNAP was not exposing this source of variation.

In addition to *Franklin*/CNL, NERSC also maintains a smaller 128 processor XT4 machine named *Silence* running CNL (*Silence*/CNL). *Silence*/CNL provided a dedicated controlled environment and allowed for direct comparisons between the two XT4 machines at a concurrency of 128. Similar behavior was observed running the hyperbolic benchmark on *Franklin*/CNL and *Silence*/CNL, which eliminated systematic effects caused by hardware configuration of *Franklin*. Additionally, after *Jaguar* was upgraded from Catamount to CNL, results nearly identical to *Franklin*/CNL were produced. This eliminated theories that the variations were unique to the *Franklin* hardware.

Other experiments that were tried include compiling with PGI and GNU compilers, using various compiler optimization flags, and running with one processor per node. In all cases the tri-normal distribution of runtimes persisted although the average runtimes were reduced with compiler optimizations and running with one processor per node.

5.2.3 Sources of Variation in Heap Management

More detailed `CrayPat` instrumentation of the memory allocation algorithm was performed which showed performance variations between 3 and 14 seconds. More importantly, there seemed to be three distinct categories of memory allocation across the processors as shown in Figure 4(a). Although the time variation in memory allocation would not account for the entire observed variation, this was compelling.

Figure 4(a) shows that the number of calls to the memory allocation routines had no correlation to the overall runtime variation. However, the total time spent in memory allocation correlated strongly to overall runtime variation, Figure 4(b), but it was a *negative* correlation: the faster the overall memory allocation, the slower the overall runtime. Looking at the data in more detail revealed that the time spent in several Fortran 77 routines, which contained no memory allocation or complex logic, was inversely correlated to the amount of time spent in the allocation and accounted for the overall runtime variation. Note that since c*alloc()* was used, the issue could not be related to the time spent mapping TLB pages since *calloc()* (unlike *malloc()*) forces the pages to be mapped. Therefore, when the memory allocator spent less time in a memory allocation operation, the performance of the Fortran numerical kernels that used the allocated memory suffered dramatically lower performance.

One hypothesis for this effect is that the CNL memory allocator has more sophisticated heuristics for managing the heap than Catamount. Simplifying the memory allocator could correct this behavior. We tested this hypothesis by changing two of the system environment variables, *MALLOC_MMAP_MAX_* and *MALLOC_TRIM_THRESHOLD_*, to simplify the memory allocation strategies. In particular the *MALLOC_MMAP_MAX_* option determines whether the *mmap()* system call will be used for large allocations. The default value is 64, but setting the value to zero eliminated expensive *mmap()* system calls altogether. The *MALLOC_TRIM_THRESHOLD_* option determines how much free space must be available on the heap (the pool of dynamically allocatable memory) before *malloc()* uses the *brk()/sbrk()* system calls to return that memory to the OS. Given we are not sharing the node with other processes, there is no good reason to return memory to the OS until we relinquish the node. Setting the *MALLOC_TRIM_THRESHOLD_* to -1 ensures that the heap only grows, and doesn't waste CPU cycles returning the memory to the OS. On the *Silence*/CNL test system, the changes to the malloc environment variables decreased the overall runtime and its variation dramatically. This did cause the time spent in memory allocation to double to an average of 25 seconds but with a variation of only one second.

The second hypothesis was that the order of memory allocation and free operations was reducing the efficiency of the data layout in the heap. Having Chombo manage its own heap could improve performance by replacing a stochastic sequence of *malloc()* and *free()* operations with a single allocation. The default memory allocation system in Chombo is *BArena* which simply makes calls to the system *malloc/free()* while tracking memory usage for memory usage diagnostics. Another Chombo memory allocation system, *CArena*, maintains several large-granularity queues of fixed sized blocks of memory in addition to tracking memory usage. When a request for new memory is made, *CArena* first checks if there is a large enough memory chunk in one of its existing queues. If so, it returns that to the user, otherwise it invokes the system *malloc()*. Freed memory is added to the queue with largest chunk-size that fits the freed memory size. The memory is never returned to the general heap. The system was originally developed for older Cray systems that behaved poorly with dynamic memory applications. Using *CArena*, the overall runtime and its variation decreased significantly and the time spent in memory allocation at any level did not increase.

Thus both hypotheses were true and both approaches appeared to be successful. The experiments were scaled up to duplicate the size of the original benchmarks. The results are shown in the two leftmost histograms in Figure 3(b), labeled "Franklin/CNL, XT4 - Environment Variables" (purple) and "Franklin/CNL, XT4 - Local Memory Management" (light blue). Clearly, the trinormal distribution disappeared when either of these optimizations were employed. The overall performance increased significantly, 15% to 20%, while the CoV decreased by a factor of three. These performance results were even better than the original Catamount results although the CoV was still greater by a factor of two.

In order to examine the performance effect of the memory allocation strategies in more detail, we measured hardware counter data using `CrayPat`. Table 1 shows a summary of the counter data of these runs collected for the PAPI_TLB_DM (Data translation lookaside buffer misses), PAPI_L1_DCA (Level 1 data cache accesses), PAPI_FP_OPS (Floating point operations) and DATA_CACHE_MISSES (Data Cache Misses) events as well as the times spent in functions.

As noted above, when the CNL malloc environment variables were set to non-default values, the Chombo memory allocation in *BArena* was about three times more expensive. However, this extra time was counterbalanced by the gains in most of the time-consuming functions, making the overall runtime shorter. This was a result of a more effective memory allocation/deallocation strategy which resulted in a large reduction in TLB misses in many key functions. For example, altering the heap allocation environment variables reduced the time spend in *getadwdxf_*, a Fortran 77 routine, from 29 to 24 seconds, mostly due to TLB misses dropping from 10.9×10^6 to 5.3×10^6. A similar effect occurred in *riemannf_*, another Fortran 77 routine. An even more dramatic effect was observed for *FArrayBox::performCopy*, in which TLB misses dropped from 8.1×10^6 to 2.4×10^6 and performance improved from 12 to 7 seconds. Also note that setting the Linux malloc environment variables to implement a more efficient memory allocation strategy significantly reduced the runtime variability of MPI processes. When Chombo's *CArena* memory management strategy was used for memory allocation and deallocation, the hardware counter statistics were similar to the statistics when the malloc environment variables were set to non-default values. As a result, the benchmark runtime was reduced and the variability of MPI processes was smaller than with the default *BArena* strategy. Therefore, we conclude that the variation was due to the more sophisticated heap management routines employed by the CNL memory management compared to the Catamount memory management. The memory allocator effects were exacerbated by the relatively small memory page sizes (4 KB) employed by CNL combined with the lack of sequential ordering preference that was provided by Catamount for page mapping.

6 Scaling Results

In this section, we describe the experiments performed on the Cray XT4 system at NCCS running Catamount, *Jaguar*/Catamount. All timing results were measured using Chombo's integral instrumentation as described in section 3.1.

The scaling experiments performed consisted of a set of runs where the number of processors was a power of two ranging from 128 to 8192. All the runs were performed with 2 levels of AMR refinement (3 levels total) and a refinement ratio of 4 between levels. The unreplicated domain was 16^3 at the coarsest level and 256^3 at the finest level. The computation was run for one coarse-level time step which, in this case, implies there were 16 time steps at the finest level. The 128 processor runs had a replication factor of 2 in the x direction and 1 in the y/z directions. Thus, the finest domain was $512\times256\times256$. As the number of processors in a run doubled, we doubled the replication in the least replicated direction starting with x then y then z. Thus, the 8192 processor runs had a replication factor of 8 in the x direction and 4 in the y/z directions and the finest domain was $2048\times1024\times1024$.

Each experiment consisted of seven runs so that we could always compute weak scaling information for each experiment. In addition, it allowed us to determine if there were anomalous or unexpected behavior at different run sizes.

6.1 Benchmark Performance

Figure 5 shows plots of normalized wall clock time for the total calculation on the *Jaguar*/Catamount, "Total" (red), and for the portion of the computation containing no communications, "Kernel" (blue). The vertical axis of the plots is normalized by the time required to solve the smallest problem on 128 processors. We observe 96% efficient scaled speedup over a range of 128 to 8192 processors, corresponding to a wall clock time of 177 ± 4 seconds to compute 2×10^9 - 1.28×10^{11} gridpoint updates. The FLOP rate for these calculations was approximately 450 Mflops/processor, which is comparatively low as a percentage of the peak processor FLOP rate, but is 85% of the peak achievable performance for the serial case using a non-adaptive computational kernel. The finest grids cover about 6.3% of the finest domain. This represents an order of magnitude improve-

Events	BArena		BArena + env vars		CArena	
	avg	CoV (%)	avg	CoV (%)	avg	CoV (%)
	getadwdxf_					
Time	29	2.09	24	0.71	26	1.65
PAPI_TLB_DM	10,943,900	9.67	5,304,520	1.30	6,681,630	2.92
PAPI_L1_DCA	27,707,000,000	0.17	27,573,100,000	0.16	27,579,600,000	0.16
PAPI_FP_OPS	1,836	1.99	1,908	1.87	1,776	2.21
DATA_CACHE_MISSES	1,021,830,000	0.72	987,418,000	0.21	989,721,000	0.20
	riemannf_					
Time	29	1.25	24	0.95	24	0.97
PAPI_TLB_DM	6,922,770	11.53	1,350,350	2.64	2,809,500	1.99
PAPI_L1_DCA	10,922,100,000	0.51	10,851,900,000	0.51	10,856,100,000	0.51
PAPI_FP_OPS	865	2.08	883	1.93	876	2.02
DATA_CACHE_MISSES	399,837,000	0.81	405,767,000	0.75	406,493,000	0.78
	FArrayBox:operator +=()					
Time	17	0.99	16	0.61	17	0.75
PAPI_TLB_DM	3,335,420	10.82	2,674,980	4.09	3,237,070	3.19
PAPI_L1_DCA	20,223,700,000	0.23	20,208,700,000	0.28	20,165,200,000	0.12
PAPI_FP_OPS	261	2.32	263	2.49	254	2.42
DATA_CACHE_MISSES	973,626,000	0.21	971,004,000	0.18	970,407,000	0.17
	secondslopediffsf_					
Time	14	6.05	4	1.47	7	1.42
PAPI_TLB_DM	5,928,940	15.59	1,328,920	1.42	2,659,650	3.01
PAPI_L1_DCA	4,525,960,000	2.14	3,486,850,000	0.16	3,488,510,000	0.16
PAPI_FP_OPS	169	5.51	408	5.64	226	3.99
DATA_CACHE_MISSES	581,476,000	6.00	204,598,000	0.16	204,569,000	0.16
	FArrayBox:performCopy()					
Time	12	4.36	7	2.14	8	1.54
PAPI_TLB_DM	8,064,020	13.23	2,385,110	1.30	2,911,920	2.11
PAPI_L1_DCA	5,902,970,000	0.18	5,846,700,000	0.18	5,868,130,000	0.21
PAPI_FP_OPS	0	0.00	0	0.00	0	0.00
DATA_CACHE_MISSES	321,141,000	0.43	333,987,000	0.16	334,268,000	0.16
	BArena:alloc()				CArena:alloc()	
Time	9	36.38	27	1.62	3	1.53
PAPI_TLB_DM	3,709,400	26.27	9,690,860	2.97	2,880,100	10.06
PAPI_L1_DCA	2,998,400,000	38.47	9,860,940,000	0.16	1,738,590,000	0.32
PAPI_FP_OPS	0	63.56	0	2.49	0	0.00
DATA_CACHE_MISSES	273,451,000	56.78	1,163,740,000	0.17	74,493,400	2.62
	GodunovPhysics:computeUpdate()					
Time	6	32.39	1	0.81	5	2.02
PAPI_TLB_DM	3,929,470	42.76	309,207	5.54	1,956,370	1.16
PAPI_L1_DCA	1,218,480,000	0.28	1,212,140,000	0.17	1,214,380,000	0.17
PAPI_FP_OPS	0	34.34	0	8.64	0	6.38
DATA_CACHE_MISSES	166,742,000	2.64	155,399,000	0.21	156,721,000	0.29
	BArena:free()				CArena:free()	
Time	5	14.67	1	1.70	1	1.60
PAPI_TLB_DM	4,619,490	14.55	321,497	8.41	816,420	14.07
PAPI_L1_DCA	389,883,000	0.55	399,047,000	0.22	545,853,000	0.32
PAPI_FP_OPS	0	0.00	0	0.00	0	26.02
DATA_CACHE_MISSES	7,349,520	3.36	5,654,840	3.37	16,148,800	3.62

Table 1. CrayPat dump of Chombo AMR code performance

ment over previous scaling demonstrations for this class of AMR algorithms [12].

7 Summary and Conclusions

The computing industry trend toward massive parallelism requires scientific application developers to dramatically improve the scalability of algorithms to stay ahead of the technology curve. In this work, we have described our experience re-architecting a production AMR application to achieve scalable performance. To help isolate scaling bottlenecks in the Chombo AMR infrastructure, we created benchmarks for an AMR hyperbolic gas dynamics computation. In so doing, we have demonstrated scalable performance with up to 8192 processors on the Cray XT4, which is the best performance to date for this class of AMR problems.

The investigation showed that the traditional concerns over load imbalance and communication volume were not as critical to application performance as identifying and isolating subtle use of non-scalable algorithms in the grid management infrastructure of the AMR framework. The investigation also revealed that variability in the system software was an equally important source of performance loss; when our AMR benchmark was run on a Cray XT4 system with CNL, it showed no performance improvement over a much slower Cray XT3 system with Catamount. However, the performance variability seen in our computations was not identified by existing OS interference diagnostics such as FWQ benchmarks and *Fixed Time Quantum* (FTQ) benchmarks [11]. Although the source of variability was eventually tracked down to differences in the heap management algorithm and implementation on these XT systems, it demonstrates that many sources of system variability are much more subtle than kernel interference and are consequently more difficult to isolate.

This work represents the first steps toward a scalable AMR code infrastructure. In the future, we will focus more attention on additional optimization, such as finer-grained load-balancing, where we see opportunities to improve scalability by another one to two orders of magnitude. Other parts of the AMR infrastructure to be studied for scalability include problem setup, regridding, data output, and checkpoint/restart of computations.

AMR offers a much more algorithmically efficient approach to scientific computing that applies computing power only where it is needed. We have demonstrated that traditional assumptions regarding the inability of AMR algorithms to scale on highly parallel computer

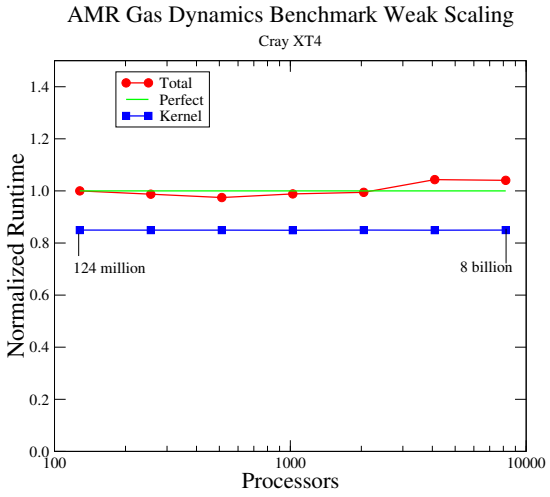


Figure 5. Hyperbolic gas dynamics weak scaling results where the y-axis is normalized to the total time for the 128 processor computation.

systems are unfounded. Ultimately, AMR frameworks such as Chombo are well poised to fully utilize petascale HPC resources that will be available in the near future.

8 Acknowledgments

We would like to individually thank: Patrick Worley for access to *Jaguar* via the PEAC INCITE grant, Helen He for her help on *Franklin* during all phases of this work, and Steve Luzmoor for his part in gathering detailed information on *Silence* and brainstorming about the results. The authors were supported by the Office of Advanced Scientific Computing Research in the Department of Energy under Contract DE-AC02-05CH11231.

References

[1] M. Berger and P. Collela. Local adaptive mesh refinement for shock hydrodynamics. *J. Computational Physics*, 82:64–84, May 1989.

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

[3] P. Colella, J. Bell, N. Keen, T. Ligocki, M. Lijewski, and B. V. Straalen. Performance and scaling of locally-structured grid methods for partial differential equations. *SciDAC*, 2007.

[4] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. V. Straalen. Chombo software package for AMR applications: design document. `http://davis.lbl.gov/apdec/designdocuments/chombodesign.pdf`.

[5] G. Johnson. P-SNAP: a system benchmark for quantifying operating system interference or noise. `http://www.c3.lanl.gov/pal/software/psnap/`.

[6] A. B. Maccabe, P. G. Bridges, R. Brightwell, R. Riesen, and T. B. Hudson. Highly configurable operating systems for ultrascale systems. In *First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters*, St. Malo, France, 2004.

[7] G. Miller and P. Colella. A conservative three-dimensional Eulerian method for coupled solid-fluid shock capturing. 183:26–82, 2002.

[8] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. *IBM*, 1966.

[9] F. Petrini, D. Kerbyson, and S. Pakin. The case of missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing*, 2003.

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

[11] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *Cluster 2004*.

[12] T. Wen, J. Su, P. Colella, K. Yelick, and N. Keen. An adaptive mesh refinement benchmark for modern parallel programming languages. In *Supercomputing*, 2007.

[13] A. M. Wissink, D. Hysom, and R. D. Hornung. Enhancing scalability of parallel structured AMR calculations. *LLNL Technical Report UCRL-JC-151791*, 2003.

[14] P. R. Woodward and P. Colella. The numerical simulation of two-dimensional fluid flow with strong shocks. 54:115–173, 1984.