A Parallel Adaptive Mesh Refinement Algorithm on the C-90 [*]

Philip Colella
Mechanical Engineering Department
University of California
Berkeley, California, 94720
and
Center for Computational Sciences and Engineering
Lawrence Livermore National Laboratory
Livermore, California 94550

William Y. Crutchfield
Center for Computational Sciences and Engineering
Lawrence Livermore National Laboratory
Livermore, California 94550

Abstract

We describe a parallel implementation of an Adaptive Mesh Refinement algorithm for gas dynamics which runs with high efficiency upon a 16 processor Cray C-90.

1 Introduction

Adaptive Mesh Refinement(AMR) is a algorithmic technique which has been shown to save as much as an order of magnitude in computational time on 3D hyperbolic systems of partial differential equations (PDE's)

[*]Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Partial support was provided by the Applied Mathematical Sciences Program of the Office of Energy Research under Contract No. W-7405-Eng-48. Partially supported by the HPCC Grand Challenge Project in Computational Fluid Dynamics and Combustion Dynamics.

[5]. AMR can be implemented very efficiently on vector architectures because it implements its adaptivity on relatively coarse grained grids which each contain many thousands of cells. However, relatively little work has been done to this point in implementing AMR algorithms upon parallel computers [1, 2]. We will describe in this paper an approach appropriate to a shared-memory architecture such as a Cray C-90.

The parallelization of an AMR algorithm offers interesting challenges. Typically the individual grid-patches of an AMR algorithm are not uniform, and indeed cover a broad span of sizes and shapes. It would therefore be quite inefficient to force the processors of a parallel computer to attack the problem in lock-step (see however the approach in [2]). The approach described in this paper avoids this problem by allowing each processor to compute asynchronously and is applicable on both dedicated and non-dedicated systems.

Implementations of AMR algorithms are typically quite complex, requiring tens of thousands of lines to express the core algorithm, completely exclusive of the user interface. It is very desirable that the parallel implementation of the AMR algorithm differ little from the original serial implementation. The reason is that all large scale AMR implementations require constant maintenance and updates. If the parallel and serial implementations are independent, this will double the code maintenance costs over the original serial implementation. This paper will describe an parallel AMR implementation technique in which less than 4% of the lines of code were modified.

In the succeeding sections of this paper, we will first briefly describe some of the principles of an AMR algorithm for hyperbolic systems of conservation laws. We will emphasize algorithmic requirements of AMR which are important to the parallel implementation. Next we will describe the implementation of the serial AMR algorithm. The serial implementation is written in a mixture of the object-oriented language C++ and FORTRAN. The choice of an object-oriented language has important implications for the parallel implementation. Next we will give an overview of the parallel implementation of an AMR algorithm for gas dynamics. We will follow with a description of a high speed I/O subsystem which is required to maintain balance between computation and I/O in the parallel AMR implementation. We will conclude with a description of a test problem which exercised the parallel AMR algorithm and achieved 5.3 gigaflops on 16 processors of a Cray C-90.

2 Adaptive Mesh Refinement Basics

Adaptive Mesh Refinement for hyperbolic systems of conservation laws is an algorithmic methodology which is useful in the solution of complex systems of hyperbolic PDE's [3]. AMR has been successfully applied to simulation of time-unsteady gas-dynamic flows in two dimensions [4] and three dimensions [5]. AMR enables the simulation of complex problems with reduced computational and storage requirements because it allows computational effort to be concentrated precisely where it is required to maintain high accuracy. Computational effort is concentrated by locally refining the computational grid. AMR algorithms manage a hierarchy of grids. The hierarchy consists of several levels of refinement. On each level n, there is a single grid spacing Δx_n, but different levels have different grid spacings related by an integer divisor. The coarsest grid covers the entire physical domain. Within that domain, rectangular subregions are covered by finer grids. Additional refinement may be achieved by recursively placing still finer grids. Finer grids take smaller time-steps which are proportional to the Δx on the grid. As time evolves, the regions of the physical domain requiring high resolution will in general change, requiring the hierarchy of coarse and finer grids to adapt dynamically to the changing solution.

Figure 1 demonstrates the hierarchical grid structure in an application of AMR to inviscid gas dynamics. The contours in Figure 1 indicate increasing density in the interaction of a shock with an inclined ramp. Each rectangle in the figure indicates an individual grid in a hierarchy of nested grids. The rectangle enclosing the entire problem domain is the single grid at the coarsest level of refinement. At the next finest level of refinement, a set of grids cover the interaction region with higher resolution. A third and finest level of refinement is shown in the figure as relatively small boxes. Note that the grids on this finest level of refinement are concentrated in regions of large gradients in the solution. This is the result of an automatic adaptation of the grids to the changing solution.

Adaptive Mesh Refinement as described in [3, 4, 5] uses a block-structured approach to local refinement. In this approach, the refined regions are not individual cells, but rather large rectangular regions with hundreds to tens of thousands of cells in each block. As a result of the block-structured approach, it is possible to amortize the overhead of managing the complex data structures which describe the region over a large regular calculation. In addition, almost all of the numerical work is done on regular arrays of floating point numbers. The amount of time spent on irregular data structures is a very small fraction of the total.

The time advancement procedure for AMR is recursive. If r is the

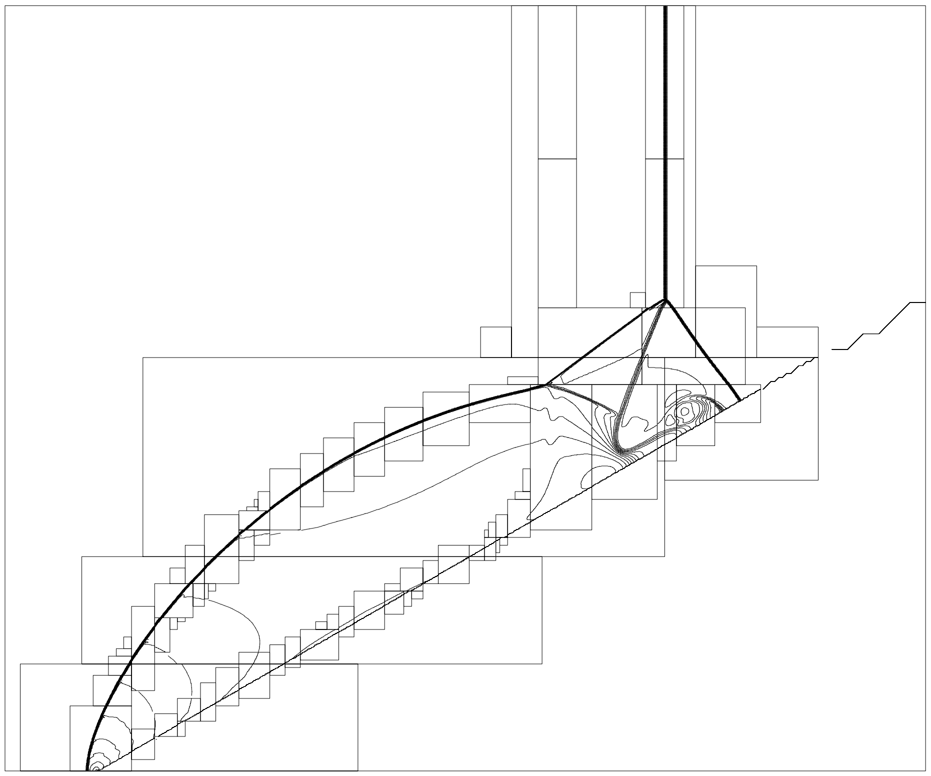

Figure 1: Application of hyperbolic AMR for inviscid compressible gas dynamics in interaction of shock and inclined ramp. Rectangles are grids used in the adaptive gridding strategy. Contours indicate changes of density. Note that grids on finest level of refinement are automatically placed on regions of large gradients in the solution.

refinement ratio between levels, $r = \Delta x_n / \Delta x_{n+1}$, then the $n+1$th level must make r time-steps for every single time-step of the n level. During the recursive evolution, the AMR algorithm must take additional steps to maintain consistency between different levels of refinement. Since error is proportional to a positive power of the grid spacing Δx, the values calculated on finer grids are always more accurate than the values of the coarser grids. Wherever fine grid cells overly coarse grid cells, the coarse grid cells are replaced with the average of the overlying fine cells at the completion of each coarse-grid time-step. An additional step must be taken to maintain conservation at the boundary between coarse and fine grids. Coarse grid cells which adjoin a fine grid but are not overlain by fine grids must be updated using fluxes which agree with the fluxes which are used on the adjoining fine grid cells. Therefore the AMR algorithm must advance these coarse grid cells using fluxes which are an appropriate sum of fine grid fluxes on the shared cell faces.

3 Serial AMR Implementation

The serial implementation of the AMR algorithm for gas dynamics comprises over 27,000 lines of code, exclusive of support libraries. Much of the complexity and most of the code of this implementation is concerned with the management of irregular data structures describing the placement of the finer grids within the coarser grids and the determination of the optimum placement of the finer grids. This part of the algorithm is not well supported by the computer language FORTRAN. However, executing these parts of the algorithm do not require a large fraction of the execution time. Most of the execution time is spent in performing floating point operations upon regular rectangular arrays. Therefore we have implemented the serial AMR algorithm for gas dynamics in a combination of FORTRAN and the object-oriented language C++ [6, 7].

The FORTRAN portion of the serial implementation is restricted to routines which operate upon single independent rectangular regions. The most complex data structure which they manipulate are multi-dimensional arrays of floating point numbers. Of course the Cray CFT77 compiler can vectorize such operations very well, with the result that the program as a whole achieves a rate of greater than 390 megaflops on a single C-90 processor. FORTRAN subroutines implement the basic finite difference time-step integration (a higher-order Godunov method [8, 9]) and utility routines such as interpolation between grids.

The C++ implementation of the AMR algorithm for systems of hyperbolic conservation laws is described in [10]. The C++ portion of the implementation consists of two parts. A C++ class library called BOXLIB supports data abstractions appropriate to rectangular block-structured algorithms. It provides abstract data types (ADT's) such as rectangular regions on a multi-dimensional integer lattice, points on an integer lattice, lists of points, floating point arrays defined on a rectangular region, etc. Through the use of these ADT's, block-structured algorithms may be described at a high level of abstraction. This improves implementer productivity, and improves clarity and maintainability.

The second part of the C++ implementation of AMR for hyperbolic systems of conservation laws consists of classes which describe the AMR algorithm. In contrast to BOXLIB, which contained classes generic to rectangular block-structured algorithms, these classes each have a direct correspondence to important parts of the AMR algorithm. An important example of this correspondence is the C++ class *Grid*. An object of class *Grid* contains all the data necessary to describe the problem solution on a rectangular region. It is convenient and reasonable to identify an object of class *Grid* with one of the rectangles shown in Figure 1. An object of class *Grid* not only contains the data describing the solution values in that region, but also information describing the placement, the time level of the data, pointers to adjoining *Grid* objects, level of refinement, etc. This encapsulation of data within a single data structure is in contrast to usual practice in FORTRAN where the data would be distributed among a number of arrays.

One particularly useful attribute of the C++ implementation is that the C++ code is almost entirely dimension independent. The same code may be compiled for one, two, or three spatial dimensions. This is a result of using high level ADT's which describe natural geometric concepts, such as rectangular regions, rather than low level data types such as arrays of integers which are naturally dimension dependent. Another useful attribute of the C++/FORTRAN implementation is that the organizational level of the implementation, written in C++, is independent of physics being simulated. The physics is totally described in the FORTRAN portion of the implementation, such as the finite difference time-step integration. This separation makes it very easy to use the same AMR superstructure for completely different physics packages.

4 Shared-memory AMR

Similar to the way that AMR utilizes a coarse-grained adaptivity, the parallel implementation utilizes a coarse-grained parallelism where the fundamental unit of parallelism is based on the C++ *Grid* class. Because *Grid* objects in AMR are relatively small, it is not very efficient to employ a do-loop approach to parallelism where many processors work on a single *Grid* object simultaneously. However, the number of *Grid* objects, particularly in 3D, may number many hundreds. There is substantial opportunity for parallelism in employing many processors to operate independently on different *Grid* objects. Any parallel overhead needed to support this decomposition may be amortized over the work needed to operate upon a *Grid* object. Because time-step advancing a *Grid* object takes a substantial fraction of a second for a single C-90 processor, the parallel inefficiency due to overhead is quite small.

Special care is required in parallel implementations to maintain parallel efficiency. High parallel efficiency is achieved by maintaining an equal distribution of work among processors. Otherwise processors will be forced to wait in idleness while an overburdened processor completes its work. Maintaining an equal distribution of work among processors is an issue when the work quanta are not uniform, as is true in AMR with *Grid* objects of greatly differing sizes. This problem is exacerbated when the parallel implementation is to be executed on a computer in non-dedicated mode. In non-dedicated mode, the parallel program will compete for computational resources with other processes. Since the results of the competition for resources cannot be predicted at compile time, it is necessary to adjust the work distribution as the parallel program runs.

The parallel implementation of AMR for hyperbolic systems of conservation laws employs a ready-queue [11] approach to load balancing. A C++ class called *ReadyQueue* maintains a list of *Grid* objects which have not been operated upon, a list of *Grid* objects which have been operated upon, and a list of *Grid* objects which are currently being operated upon. Each processor iterates in a loop where it takes *Grid* objects from the ready-list, operates upon them, and returns them to the finished list. Locks [11] prevent different processors from manipulating the lists simultaneously. Different processors are able to operate upon different *Grid* objects simultaneously. Figure 2 illustrates this cycle.

The C++ class *ReadyQueue* is constructed so that the operation of ready queues is very similar to iterating across *Grid* objects in the serial implementation. The *ReadyQueue* class takes a list of *Grid* objects and a pointer to a function which operates upon *Grid* objects as arguments. It then iterates across the *Grid* objects in parallel, applying the function to each. As a concrete example, consider a SERIAL loop which would be written as follows:

```
GridList gl[MAXLEV];
int level;
void advance( Grid *);
Grid *gptr;
     .
     .
for( gptr=gl[level].first();
     gptr != NULL;
     gptr = gl[level].next(gptr) ){
          gptr->advance();
```

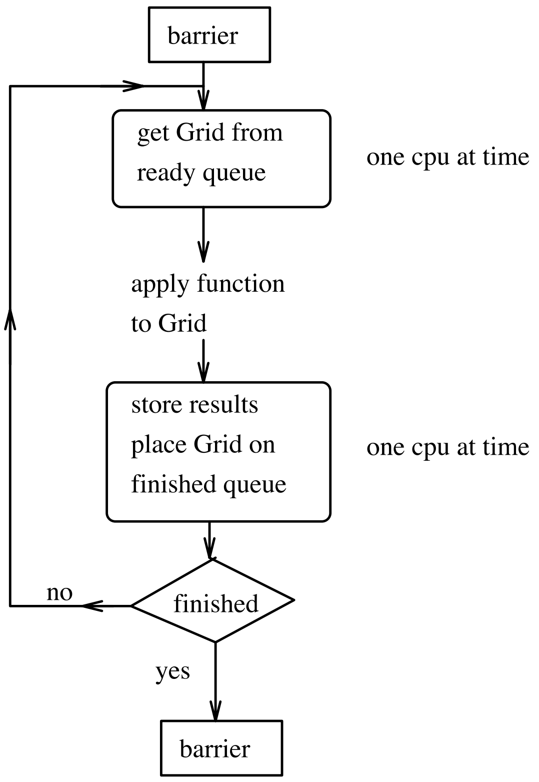

Figure 2: Basic iteration cycle of each processor in Ready Queue.

```
}
```

This code fragment applies the function *advance* to each *Grid* object in one level of refinement. The same code in the PARALLEL implementation is very similar:

```
GridQ *gq;
gq = new GridQ(NTASKS);
    .
    .
GridList gl[MAXLEV];
int level;
void advance( Grid *);

gq->setup( gl[level],advance);
gq->doParallel();
```

The encapsulation of important data within the *Grid* class facilitates the synchronization of parallel operations upon the *Grid* objects. A major problem in parallel implementations is to properly synchronize the processors in their access to data in order to prevent processors from interfering with each other. Because a *Grid* object contains almost all of the information necessary to perform operations upon it, allowing only a single processor to operate upon a given *Grid* object obviates most of the synchronization problems. However, not all operations can be performed with just the data contained within a *Grid* object. For example, to time-step advance a *Grid* object requires boundary data which resides upon adjacent *Grid* objects. We therefore also require that some kinds of read-only data be available from *Grid* objects and that a *Grid* object not modify such data during a parallel operation. The parallel AMR implementation described in this paper has not required further synchronization beyond this requirement and the access-locks described for the *ReadyQueue* class.

All parallelism resides in the C++ portions of the implementation. FORTRAN portions only operate upon single rectangular regions and are therefore unaware of parallelism across *Grid* objects. This maintains the independence of the physics packages, written in FORTRAN, from the organizational levels of the AMR implementation and facilitates easy replacement of physics packages. This also makes it easier to develop physics packages on serial architectures, including workstations, and then realize parallel performance without further development upon the physics package. The parallel AMR implementation has only four parallel sections:

1. time-step integration of all the *Grid* objects of a single level of refinement.
2. enforcement of consistency between coarse grids and fine grids.
3. Richardson error estimation during calculation of optimum grid positions.
4. initialization of new *Grid* objects from data in old *Grid* objects during calculation of optimum grid positions.

In all, less than one thousand lines were changed in the 27 thousand line serial implementation to make it parallel. This includes totally new classes which were added to manage the ready-queue and high-speed I/O, which is described in the next section.

5 High Speed Output for Parallel AMR

In parallel implementations which run on dedicated or nearly-dedicated computers, high performance output can be quite important. If a computer is shared with many processes, when one process is writing results to disk, another process can utilize the CPU. However, if the computer is dedicated or nearly dedicated, time spent in output is lost. To give some figures typical of the parallel AMR implementation: a program will run for 9300 wall-clock seconds and output 8.1 gigabytes of data. If the data were written to disk at an average rate of 8 megabytes per second, this would take 1012 seconds. This would cause the parallel computer to spend more than 10% of the time with no processors performing calculations. This shows the necessity of providing high speed output mechanisms for parallel programs which output large amounts of data.

For the parallel AMR implementation, we have implemented an output method which utilizes the Solid State Disk of the NERSC Cray C-90 to asynchronously write data to disk while the CPU performs computations [12]. Figure 3 is a diagram showing interconnection of CPU to I/O devices on the C-90. The high speed output is performed as follows. When the CPU writes data, instead of writing directly to disk, it writes to buffers allocated on the SSD through a 2 gigabyte per second channel. After the CPU is finished writing the data, it returns to computation. While the CPU computes, the data is written from the SSD to the I/O Cluster via a high speed channel which by-passes the CPU. The I/O Cluster in turn writes the data to disk. From the point of view of the CPU, the time expended writing the output was only the time it takes to write to the SSD. In practice, when writing program state to disk in large parallel runs, we have reached speeds of 300 megabytes per second effective speed, approximately 30 times the speed of writing directly to slow disk.

As described previously, we prefer to minimize the difference between the parallel implementation and the serial implementation of AMR for hyperbolic systems of conservation laws. Because output is done in many places in the serial implementation, we have created a C++ class called *SDSstream* which manages the high speed output. The *SDSstream* class is derived from the standard C++ output class, *ostream*. Since it is a derived class of *ostream*, every function which takes an *ostream* as an argument will accept a *SDSstream*. It is not necessary to modify large numbers of functions to allow them to use the high speed output class. The C++ class *SDSstream* performs 5 functions:

1. allocate large buffers of the SSD.

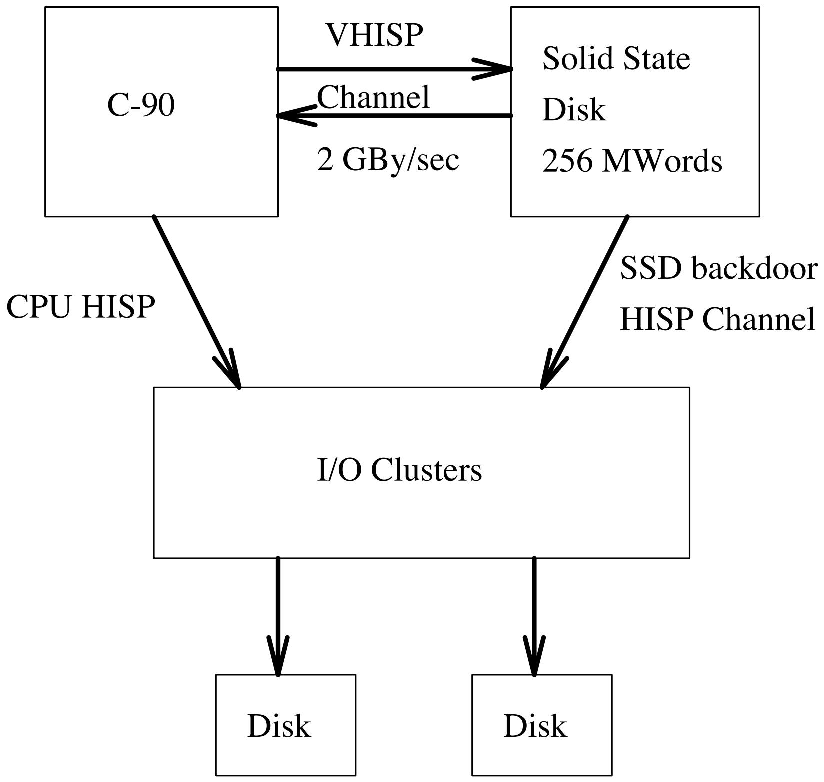

Figure 3: Functional diagram showing interconnection of CPU, SSD, and I/O devices on NERSC C-90.

2. writes data to buffers on SSD.
3. initiates asynchronous write on buffers when full.
4. monitors status of asynchronous writes and reuses buffer when write is complete.
5. closes files when all buffers have been written.

6 Example

We have utilized the parallel implementation of AMR for hyperbolic systems of conservation laws in a large scale simulation of inviscid gas dynamics in a 3D spatially evolving mixing layer. Figure 4 shows a side view of the physical domain. Two streams of gas enter on the right hand side separated by a splitter plate. One stream is super-

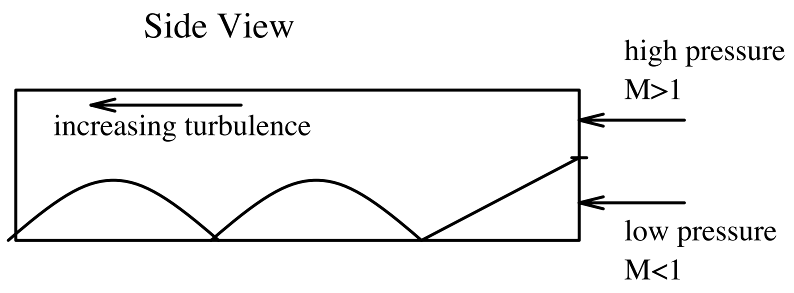

Figure 4: Side view of physical domain in 3D spatially evolving mixing layer computation.

sonic and at high pressure while the second is sub-sonic and at low pressure. A stationary shock forms at the inlet which is trapped in the low pressure layer as it flows downstream. The trapped shock sets up a cellular structure with primary and secondary instabilities. As the gas flows downstream, the 3D secondary instabilities begin to dominate the flow and eventually transition the flow to 3D turbulent flow. Figure 5 is a perspective rendering of density at one time instant showing transition to turbulence. On the finest level of refinement, the computational space covers a region 1040 by 104 by 64 cells wide.

In one of the production runs performed on this problem, 9300 wall-clock seconds were expended on the 16 processor NERSC C-90. During this time, 8.1 gigabytes of data were written in 307 files. The run utilized 128,209 CPU seconds and achieved a rate of 5.293 gigaflops, as measured by the hardware performance monitor. This includes the time spent doing I/O at the very beginning and end of the run, where overlapping I/O with calculation is not possible. Peak rates exclusive of the beginning and end would be higher.

Figure 6 shows the amount of time during the computation as a function of number of CPU's working on the calculation. Several points should be noted in Figure 6. First, although the time spent in each processor-state increases quickly as the number of processors increases, the time spent with 16 processors is not greatly different from 15 processors. This calculation was performed on the NERSC C-90 in nearly-dedicated mode. In nearly-dedicated mode, all of the batch queues were stopped. However, users were still able to run programs in foreground mode on the C-90. Because the Special Parallel Programming Allocation (SPPA) runs were made at night and on weekends, the number of foreground users was small, but it was not zero. We have observed in some shorter runs that the processor usage peaks sharply at 16 processors. Second, there is a perceptible peak in this

Figure 5: 3D perspective rendering of density in spatially evolving mixing layer.

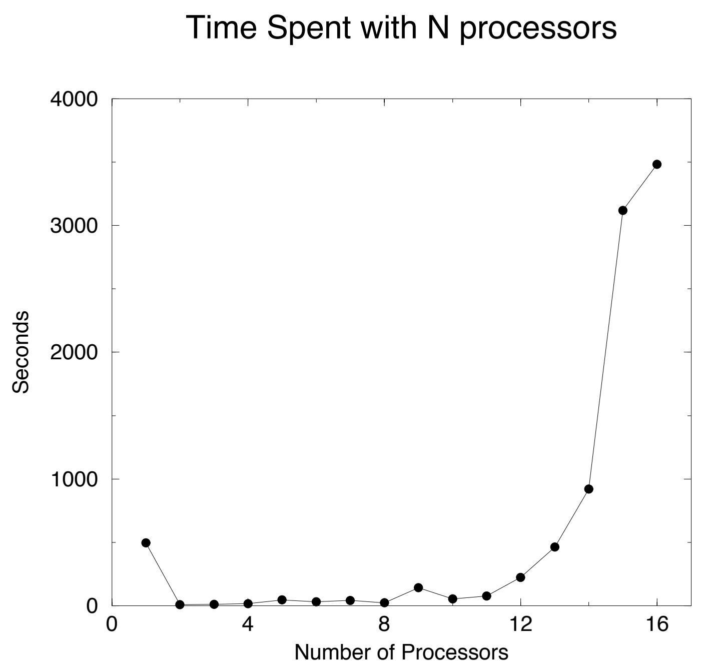

Figure 6: Time spent in calculation with N processors working on calculation.

graph at 9 processors. This calculation utilized more than a hundred grids on the finest level, but only 9 at the coarsest level. Calculations on the coarsest level would then contribute to the 9 processor state. Lastly, the time spent with one processor indicates all non-parallel time spent in execution, including the time managing high-speed I/O. On the average, 13.85 processors were active on the computation.

7 Conclusions

Adaptive Mesh Refinement(AMR) is a algorithmic technique which has been shown to save as much as an order of magnitude in computational time on 3D hyperbolic systems of partial differential equations (PDE's) [5]. The results of this paper demonstrate that AMR can run on parallel shared memory architectures with high efficiency. Furthermore, due to the organization we had previously imposed upon our serial implementation of AMR for hyperbolic systems of conservation laws, we were able to implement parallel AMR with very little new code. Our object-oriented implementation was a major cause of this result. We have also used object-oriented techniques to encapsulate a special parts of the parallel implementation, such as ready-queue maintenance and high-speed output.

8 Acknowledgments

This work was performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Partial support was provided by the Applied Mathematical Sciences Program of the Office of Energy Research under Contract No. W-7405-Eng-48. Computations were carried out on the National Energy Research Supercomputing Center's Cray C-90 under the Special Parallel Processing Allocation. Steve Luzmoor of Cray Research Inc. aided in the high speed I/O implementation.

References

[1] W. Y. Crutchfield, "Load Balancing Irregular Algorithms", UCRL-JC-107679, July 1991.

[2] M. J. Berger and J. Saltzman, "AMR on the CM-2", Applied Numerical Math. 14, 239-253, 1994.

[3] M. J. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations," J. Comp. Phys., vol. 53, pp. 484-512, 1984.

[4] M. J. Berger and P. Colella " Local Adaptive Mesh Refinement for Shock Hydrodynamics", J. Comp. Phys., vol. 82, pp. 64-84, May 1989.

[5] J. B. Bell, M. Berger, J. Saltzman, and M. Welcome, "Three Dimensional Adaptive Mesh Refinement for Hyperbolic Conservation Laws", UCRL-JC-108794, to appear in SIAM J. Sci. Stat. Computing.

[6] S. B. Lippman, "C++ Primer,", Addison-Wesley Publishing Company, 1989.

[7] B. Stroustrup, "The C++ Programming Language," Addison-Wesley Publishing Company, 1986.

[8] P. Colella, "A Direct Eulerian MUSCL Scheme for Gas Dynamics," SIAM J. Sci. Stat. Comput., vol. 6, p. 104, 1985.

[9] P. Colella and P. R. Woodward,"The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations," J. Comp. Phys., vol. 54, pp 174-201, 1984.

[10] W. Y. Crutchfield and M. L. Welcome, “Object-Oriented Implementation of Adaptive Mesh Refinement Algorithms”, Scientific Programming, vol. 2, pp. 145-156, Winter 1993.

[11] “Cray Y-MP, Cray X-MP EA, and Cray X-MP Multitasking Programmer’s Manual”, publication SR-0222 F-01, Cray Research Inc., 1989.

[12] This method was shown to us by Steve Luzmoor of Cray Research Inc.