Adaptive Mesh Refinement in Titanium

Tong Wen and Phillip Colella
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
{ twen, colella }@lbl.gov

Abstract

In this paper, we evaluate Titanium's usability as a high-level parallel programming language through a case study, where we implement a subset of Chombo's functionality in Titanium. Chombo is a software package applying the Adaptive Mesh Refinement methodology to numerical Partial Differential Equations at the production level. In Chombo, the library approach is used to parallel programming (C++ and Fortran, with MPI), whereas Titanium is a Java dialect designed for high-performance scientific computing. The performance of our implementation is studied and compared with that of Chombo in solving Poisson's equation based on two grid configurations from a real application. Also provided are the counts of lines of code from both sides.

1. Introduction

Since first developed by Berger and Oliger [6] for hyperbolic Partial Differential Equations (PDEs), the Adaptive Mesh Refinement (AMR) methodology has been successfully applied to numerical modeling of a variety of physical problems that exhibit multiscale behavior ([7] [4] [2] $\cdots$). Here, the multiscale behavior is characterized by localized large derivatives of the solution. For example, solutions to Poisson's equation can have this type of variation due to local concentration of charges or boundary conditions.

In finite difference calculations, truncation errors are in the form of

$$(\Delta x)^p M(\phi),$$

where Δx is the mesh size, $M(\cdot)$ is a $(p+q)$-order differential operator, and ϕ is the solution. If ϕ is not smooth, then a small Δx is required so as to resolve the solution in the necessary accuracy. For the above class of problems, uniformly refining the mesh to meet the resolution requirement would be a waste of computational resources, because high resolution is not needed everywhere. In AMR, computational effort is adjusted locally to maintain a uniform level of accuracy throughout the problem domain. That is, areas of interest are covered with finer grid patches than the surrounding regions. For time-dependent problems, the finer meshes are also advanced with a smaller time step. By saving computational resources, AMR allows bigger and harder problems to be attacked.

The idea of AMR sounds straightforward, but it is challenging to implement it efficiently. One aspect of the difficulty comes from the irregular data access and computation in AMR algorithms. In our approach to AMR, regions at the same refinement level are covered by a union of rectangular grids. Here, the refinement level l ranges from 0 (coarsest) to $l_{\max}$ (finest). Grids at level $l+1$ are embedded recursively inside the union of the grids at level l. Having such a nested hierarchy of grids introduces interfaces not only between grids at the same refinement level but also between those from two adjacent levels. Note that in practice there can be thousands of grids at one level. Matching the numerical solution at these artificial boundaries is the source of the irregular data access and computation, and parallelizing these irregular operations is especially challenging.

As a simulation goes on, the resolution requirement for the evolving solution may change from time to time. If there is such a change, grids are regenerated accordingly. Reconstructing the grid hierarchy is not trivial when AMR computation is performed in parallel, because load balancing has to be done at runtime. AMR programs are hard to write not only because of the challenges described above, but also because complicated are the control structures of AMR iterations and the interactions between levels of refinement. In this paper, as an example we introduce an AMR multigrid algorithm for second-order elliptic PDEs.

In order to exploit modern software design principles, particularly object-oriented programming, two AMR software packages CCSE Applications Suite and Chombo have been developed in both C++ and Fortran at Lawrence Berkeley National Laboratory (LBNL), which has been at the forefront of AMR-algorithm designs and applications. The benefit of combining these two programming languages is obvious. Firstly, multidimensional arrays are built-in types in Fortran, while they must be provided as a class library in C++. Although the performance of C++ programs has been improved during the last decade, it is still hard for a C++ multidimensional array library to outperform Fortran. Overall, performance is the primary goal here. On the other hand, Fortran does not have some features that C++ has such as template and inheritance, which provide tremendous potential for code reuse and generic programming. In the above two implementations, the regular operations on arrays are written in Fortran, while the irregular ones are written in C++. However, the price to pay for using a mixture of programming languages is that code maintenance is high and debugging is difficult.

New languages have been developed to challenge the popular library approach to parallel programming. Among them is Titanium, a Java dialect designed for high-performance scientific computing [32]. Titanium supports multidimensional arrays and an explicitly parallel SPMD (single program, multiple data) model of computation with a global address space. Using Java as its base makes Titanium easy to learn, meanwhile providing users with modern programming technologies. Titanium compiler is designed to translate Titanium into C for portability and economy. With the global address space, data residing on different processors can be accessed transparently by Titanium processes. In Titanium, one-sided communications are supported at language level. Titanium programs run on both shared-memory and distributed-memory architectures.

The goal of this paper is to evaluate through a case study Titanium's usability as a parallel programming language for scientific applications. Following Chombo's design of software architecture, we build in Titanium the infrastructure for AMR applications and an elliptic PDE solver above it. Then, our implementation is compared with its Chombo counterpart in both performance and expressiveness.

2. A case study: an AMR elliptic PDE solver in Titanium

As a case study of Titanium's usability, we implement a subset of Chombo in Titanium, which includes basic AMR data structures and operations, and a solver for elliptic PDEs. The reason to choose the elliptic solver is that this class of problems, particularly Poisson's equation, are among the most common PDEs encountered in various areas of science and engineering. Meanwhile, its implementation represents the difficulties described above. We follow Chombo's design of software architecture with modifications to suit Titanium.

2.1. Basic data structures and operations

In this paper, we are interested in solving equations

$$L\phi = f \text{ in } \Omega, \tag{2.1}$$

where L is an elliptic partial differential operator and Ω is a bounded, open domain. Particularly in our test problems, L is the Laplacian operator, that is, $L = \Delta$, subject to Dirichlet boundary condition

$$\phi = g \text{ on } \partial\Omega. \tag{2.2}$$

As a matter of notation, for each refinement level l, we denote by Ω^l the union of rectangular grids at this level. Let ϕ^l and f^l be the discretized versions of ϕ and f defined on Ω^l. In this paper, superscripts are exclusively used as indexes of refinement levels. Thus, L^l is the discretized version of L at level l. Note that as explained later, the evaluation of L^l for instance on ϕ^l involves not only ϕ^l itself but also ϕ^{l-1} and ϕ^{l+1} (assuming $0 < l < l_{\max}$).

The most fundamental data structures in our implementation are a metadata class and a data class. The metadata class contains the set of rectangular grids at one refinement level along with their processor assignments. In Titanium, a rectangular grid is represented by a built-in type *RectDomain*, and another built-in type *Point* is used to index each cell in a grid. In Figure 1, an example of two adjacent levels of grids is given for the two dimensional case. In this example, an object of the metadata class for the coarse level contains two *RectDomain*s, while one for the fine level contains three. The metadata class also have methods that work on the grids it owns, such as *coarsen* and *refine*.

The data class is defined on the metadata class. It contains the data built on the blueprint provided by the metadata class. Typically, an object of the data class contains a union of multidimensional Titanium arrays constructed on the grids that are described in its defining metadata-class object, and these arrays are distributed according to the processor assignments. Each local copy of the object has transparent access to all the arrays it has no matter where they live. As mentioned before, Titanium supports multidimensional arrays as built-in types. In our implementation, for example, ϕ^l and f^l are objects of the data class.

Both physical and artificial boundary conditions are dealt with ghost cells. The ghost cells of a grid are a layer of cells surrounding it. The size of this layer depends on what numerical schemes are used. In this paper, its size is one everywhere. For data-class objects that are operands of the discretized operators L^l, they must have ghost cells. An important method of the data class is *exchange*, which exchanges

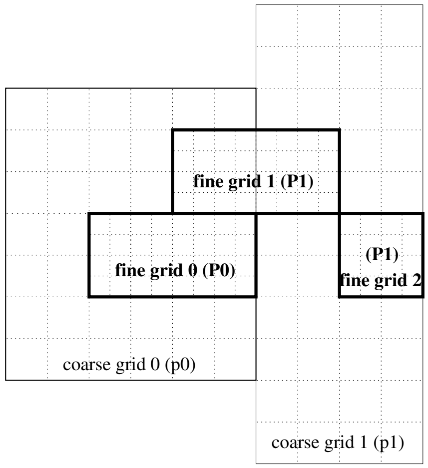


Figure 1. Two adjacent levels of grids are shown in this example for the two dimensional case. There are two coarse grids at the coarse level and three fine grids at the fine level. For each grid, the number in parenthesis represents the processor it is assigned to. The dotted squares in each grid are the cells it contains. The refinement ratio between these two levels is 2. In this paper, all AMR algorithms are cell-centered.

the values of two adjacent arrays (at the same refinement level) at their grid boundary by copying them to the corresponding ghost cells. This copy operation is performed by the *copy* method of Titanium array. Given two Titanium arrays TA_1 and TA_2, $TA_1.copy(TA_2)$ copies the contents of the elements of TA_2 into the elements of TA_1 that have the same indexes, where TA_1 and TA_2 can belong to different processors. For our Poisson solver, a large portion of the communication time is consumed by the *exchange* method.

The basic AMR data structures are implemented in 2000 lines of Titanium code in contrast to around 35000 lines of code in Chombo. Note that we do not include comments in the lines of code. Even after consideration of the simplifications we have made and the limitation of the lines of code as a productivity measure, there is still a significant save in programming effort for using Titanium.

Frequently used operations on data-class objects are implemented as methods of classes. Some of these operations are used to match the solution at the grid interfaces between two adjacent refinement levels, while others are used to pass information from one level to another. One method we want to mention is *CFInterp*, which determines the values at ghost cells with quadratic interpretations. This operation is a good example of the ones that involve both irregular data access and computation. The basic AMR operations are implemented in 1200 lines of Titanium code, whereas the corresponding part in Chombo has around 6500 lines of code.

2.2. An AMR multigrid algorithm

We use the same multigrid algorithm for elliptic problems as Chombo does, which is described at high level in Figures 2 to 4. The goal here is to provide readers an overall picture of this algorithm and establish the notations for our later discussions. For its details, please refer to [21] and Chombo's design documents available at its website [12].

Note that the meaningful parts of ϕ^l for $l = 0, \cdots, l_{\max}$ are those whose domains are not covered by any finer grids. They together serve as the numerical solution to (2.1) on the grid hierarchy $\Omega^0, \cdots, \Omega^{l_{\max}}$. Hereafter, we denote this composite solution by ϕ^{comp}, and let L^{comp} be the corresponding discretized operator.

At each level l, on the inner regions of Ω^l that are away from $\partial\Omega^{l+1}$ (if it exists) L^l is simply the usual $(2N+1)$-point stencil, where N is the space dimension. In the boundary regions (both $\partial\Omega^l$ and $\partial\Omega^{l+1}$) L^l addresses the various boundary conditions using the corresponding ghost cells. Since these ghost cells may be filled in with additional information from either level $(l-1)$ or level $(l+1)$, at most three levels of data are needed to evaluate L^l on Ω^l. In the multigrid algorithm described here, a simplified version of L^l denoted by L^l_{nf} is also used, where it is assumed that there is no finer levels above l.

To illustrate how the multigrid algorithm iterates through the refinement levels, shown in Figure 5 is a simple sketch of one multigrid V-cycle on a three-level grid hierarchy. The iteration starts at the finest level on the way down to the coarsest level. After reaching the bottom, it goes back to the top level. The arrows in this diagram show how information flows through refinement levels. Note that the small V-cycles are from procedure mgRelax, which introduces intermediate refinement levels. This AMR multigrid algorithm for Poisson's equation is implemented in 1500 lines of Titanium code, where 90% of our code is reusable for other elliptic PDE solvers.

3. Test problems and preliminary profiling results

Our implementation has covered almost all Titanium's features including those added to Java such as templates, immutable classes, and zone-based memory management. In this section, we study the performance of our elliptic solver in solving Poisson's equation on a cubic domain subject to

```
procedure AMRSolve():
R^comp := f^comp − L^comp(φ^comp)
while (||R^comp|| > ε||f^comp||)
      AMRVCycle(l_max)
      R^comp := f^comp − L^comp(φ^comp)
end while

procedure AMRVCycle(level l):
if (l = l_max) then R^l := f^l − L^l_nf(φ^l, φ^{l−1})
if (l > 0) then
      φ^l_copy := φ^l on Ω^l
      e^l := 0 on Ω^l
      mgRelax(e^l, R^l, r^l)
      φ^l := φ^l + e^l
      e^{l−1} := 0 on Ω^{l−1}
      R^{l−1} := Average(R^l − L^l_nf(e^l, e^{l−1})) on C_{r^l}(Ω^l)
      R^{l−1} := f^{l−1} − L^{l−1}(φ^comp) on Ω^{l−1} − C_{r^l}(Ω^l)
      AMRVCycle(l − 1)
      e^l := e^l + Interpolate(e^{l−1})
      R^l := R^l − L^l_nf(e^l, e^{l−1})
      δe^l := 0 on Ω^l
      mgRelax(δe^l, R^l, r^l)
      e^l := e^l + δe^l
      φ^l := φ^l_copy + e^l
else
      solve L^0_nf(e^0) = R^0 on Ω^0.
      φ^0 := φ^0 + e^0
end if
```

Figure 2. Pseudo-code description of the AMR multigrid algorithm. Given φ^{l-1} and φ^{l}, $Average(\varphi^l)$ passes the value of φ^l down to refinement level $(l-1)$ via averaging, while $Interpolate(\varphi^{l-1})$ passes the value of φ^{l-1} up to refinement level l via piecewise constant interpolation. Here, $\mathcal{C}_{r^l}(\Omega^l)$ coarsens the grids in Ω^l down to level $(l-1)$ by a ratio of r^l.

```
procedure mgRelax(φ^f, R^f, r)
{
      for i = 1, ..., NumSmoothDown
            GSRB(φ^f, R^f)
      end for
      if (r > 2) then
            δ^c := 0
            R^c := Average(R^f − L^f_nf(φ^f, φ^c ≡ 0))
            mgRelax(δ^c, R^c, r/2)
            φ^f := φ^f + Interpolate(δ^c)
            for i = 1, ..., NumSmoothUp
                  GSRB(φ^f, R^f)
            end for
      end if
}
```

Figure 3. Recursive relaxation procedure. Here the refinement ratio between level f and level c is 2.

```
procedure GSRB(φ^f, R^f)
{
      φ^f := φ^f + λ(L^f_nf(φ^f, φ^c ≡ 0) − R^f) on Ω^f_BLACK
      φ^f := φ^f + λ(L^f_nf(φ^f, φ^c ≡ 0) − R^f) on Ω^f_RED
}
```

Figure 4. Gauss-Seidel relaxation with red-black ordering. Here λ is the relaxation parameter.

Dirichlet boundary condition, that is,

$$\begin{cases} \Delta\phi & = f \text{ on } \Omega, \\ \phi & = g \text{ in } \partial\Omega. \end{cases}$$

This problem is discretized using two grid configurations from a real application. Some of the timing results are compared against those of Chombo, which we use as benchmarks. Our study is based on two platforms where the same version of code is used and compiled with the Titanium compiler of version 2.573.

The two grid configurations are described in Figure 6, which are from an incompressible fluid flow problem [23]. The small configuration is for the case where the problem domain contains two vortex rings, while the large one is for the case where there is one vortex ring. We use the small configuration to test the serial performance of our solver and the large one to test its parallel performance.

To make it easy to compare with Chombo, we set $f = 0$, $g = 0$, and the initial guess $\phi^{\text{comp}}_{\text{init}} = 1$ for the two test cases. But, this simplification does not reduce the generality of the numerical computations involved. We choose the convergence criteria to be

$$||R^{\text{comp}}||_\infty \le 10^{-10}||R^{\text{comp}}_{\text{init}}||_\infty.$$

The residual norms from each iteration are listed in Appendix. Note that not all the operations in the Titanium code are implemented in the same way as Chombo does. For example, at the base level the two sides use different numerical methods to solve the problem

$$L^0_{\text{nf}}(e^0) = R^0 \text{ on } \Omega^0,$$

which however is not performance critical. Therefore, we do not expect that the residual norms from both sides match exactly.

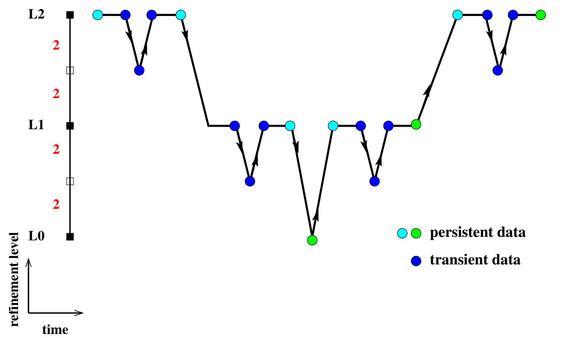
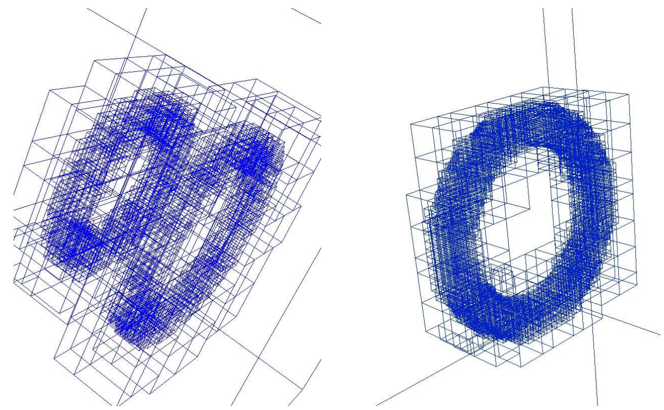


Figure 5. A simple sketch of one multigrid V-cycle. There are three levels of refinement in this grid hierarchy. The refinement ratio is 4 between two adjacent levels. Here, the direction of data flow is indicated by arrows. The small V-cycles are contributed by the mgRelax subroutine, which itself is a multigrid algorithm.



the small configuration			the large configuration		
level	number of grids	number of cells	level	number of grids	number of cells
0	1	32768	0	64	2097152
1	106	279552	1	129	3076096
2	1449	2944512	2	3159	61468672

Figure 6. Two grid configurations. Each box here represents a three dimensional grid. At the base level, the union of the grids fully cover the problem domain. There are totally 32^3 and 128^3 cells at the base levels of the small and large configurations respectively. The refinement ratios between two adjacent levels are 4 for both cases.

To separate the computation-intensive part of AMRSolve from those that involve either communication or irregular computation, we do not include in GSRB the operations that fill in the ghost cells. Most of these irregular operations are performed by the *exchange* and *CFInterp* methods, which are timed separately. The *CFInterp* method does two kinds of calculation, one does not involve communication while the other one does. Hereafter, we denote the first case by CFInterp1 and the second one by CFInterp2. All the timing results in this paper are in seconds. In the parallel mode, the maximum running time of each operation is reported.

The serial performance of our solver is tested on an Intel Pentium 4 workstation. The timing results from both sides along with flop counts are listed in Table 1. The mesh size is $1/32$ at the base level. We can see that the serial performance of the Titanium solver (10% faster) matches very well that of Chombo.

The parallel performance is studied on an IBM SP RS/6000 supercomputer named Seaborg (`http://www.nersc.gov/nusers/resources/SP/`). First, we run the small test problem within one node using different number of processors. The timing results and the speedup factors are shown in Figure 7. We can see that the Titanium solver scales almost linearly within one node on Seaborg. The scalability of GSRB is perfect because all the computations involved are regular and local. Since *exchange* is communication intensive, it does not scale as well as GSRB.

The large test problem is then distributed across multiple nodes. On each node, 14 processors are used. The timing results are listed in Table 2. Here, the mesh size is $1/128$ at the base level. We can see that the *exchange* method does not scale with the number of nodes. It becomes the bottleneck of the overall scalability when more than two nodes are used. The reason is that the source and destination regions of *exchange* are non-contiguous in linear storage. This fact leads to high overhead of Titanium array's *copy* method used in both *exchange* and CFInterp2 for communication. One possible solution is to implement a new *exchange* method that does packing and unpacking as Chombo does to reduce the cost of communication. Another way is to improve the performance of the *copy* method. People at Titanium group is investigating more efficient packing in the GASNet communication system in order to achieve this goal.

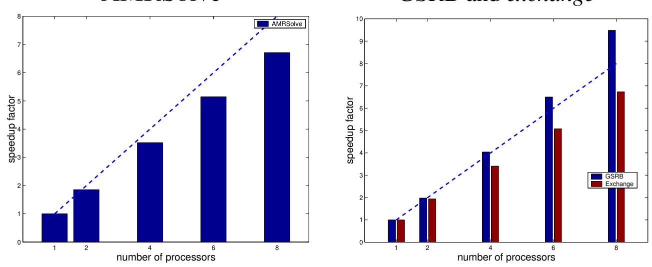
In Table 3, we compare the two solvers for the case where two nodes are used. The first column of this table is copied from Table 2. For this case, the Titanium AMRSolve (16% slower) is still able to match its Chombo counterpart. One observation here is that the Titanium GSRB is significantly

	Titanium	Chombo
AMRSolve	**52.15**	**57.47**
GSRB	12.98	11.64
exchange	11.25	17.31
CFInterp1	5.91	4.19
CFInterp2	4.97	4.31
flops (billion)	9.45	10.01

Table 1. Serial performance. This comparison is conducted on an Intel Pentium 4 (2.8 GHz) workstation based on the small test problems. The timing results are in seconds.

the scalability of AMRSolve

the scalability of GSRB and *exchange*



number of processors	1	2	4	6	8
AMRSolve	**138.4**	**74.63**	**39.31**	**26.90**	**20.62**
GSRB	33.85	17.11	8.38	5.21	3.57
exchange	27.66	14.26	8.12	5.45	4.11
CFInterp1	14.33	7.13	2.88	1.51	1.00
CFInterp2	15.27	9.33	4.90	3.41	2.89

Figure 7. The scalability of the small test problem on Seaborg. The timing results are in seconds.

slower than the Chombo GSRB which is implemented in Fortran. This difference in performance is related to the different orders in which the two GSRBs iterate through the red/black cells during the GSRB update. It turns out that the Chombo GSRB ordering is more efficient, because it provides better locality of the data. In the second column of this table, listed are the timing results of the Titanium code where in GSRB the red/black cells are updated in the same order as in Chombo. By doing so, the performance of the Titanium GSRB is improved by 14%, which now is close to that of its Chombo counterpart.

number of nodes	1	2	3
AMRSolve	204.7	134.6	116.7
GSRB	58.39	29.46	19.18
exchange	**42.60**	**41.32**	**46.23**
CFInterp1	10.05	5.20	3.78
CFInterp2	12.53	10.03	9.87

Table 2. The scalability of the large test problem on Seaborg. On each node, 14 processors are used. Here, the Titanium code is compiled in 64-bit mode. The timing results are in seconds.

	Titanium		Chombo
	old GSRB ordering	Chombo GSRB ordering	
AMRSolve	134.6	130.0	113.3
GSRB	**29.46**	**25.34**	**22.71**
exchange	41.32	40.56	37.12
CFInterp1	5.20	5.15	6.17
CFInterp2	9.87	10.10	7.97
flops (billion)	172.9		171.0

Table 3. Parallel performance. This comparison is conducted on Seaborg based on the large test problem, where two nodes and totally 28 processors are used. The first column of this table is copied from Table 2. In the second column, listed are timing results where in the Titanium GSRB the red/black cells are updated in the same order as in Chombo. Here, the Titanium code is compiled in 64-bit mode. The timing results are in seconds.

4. Conclusions and future work

Titanium is a high-level parallel programing language for scientific applications, which supports modern software design principles. It is easy to learn and to use this language. From this case study, we can see that writing AMR applications in Titanium requires much less programming effort than in other languages. Meanwhile, Titanium has potential to provide high performance for this class of applications, which is comparable with that of production-level packages such as Chombo.

As a product of this project, several improvements have been done to Titanium. Now, templates are fully supported, and a more efficient rectangular-domain library has been developed. Our next task is to improve the scalability of *ex-*

the small test problem		
iteration	Titanium	Chombo
initial	6144.0	6144.0
1	0.2727	0.2728
2	0.2538	0.2538
3	0.0091	0.0092
4	3.706E-04	3.580E-04
5	5.093E-06	4.748E-06
6	2.090E-07	1.570E-07

the large test problem		
iteration	Titanium	Chombo
initial	9.830E04	9.830E04
1	4.169	4.169
2	1.290	1.277
3	0.0222	0.0219
4	1.046E-03	1.039E-03
5	1.761E-05	2.128E-05
6	7.893E-07	7.367E-07

Table 4. The infinity norms of the residuals from the two test problems.

change. A more detailed comparison between our Titanium implementation and its Chombo counterpart is also interesting. The ultimate goal is to provide an environment in Titanium where high-performance AMR applications can be developed easily.

Acknowledgments

This project is supported by Lawrence Berkeley National Laboratory. Our thanks go to all the members of Titanium group at University of California, Berkeley for their support and weekly discussions. Among them, we especially want to thank Dan Bonachea, Jimmy Su, and Amir Kamil for their valuable suggestions. We also want to thank Noel Keen at Lawrence Berkeley National Laboratory for providing profiling results of Chombo.

Appendix

In Table 4, listed are the infinity norms of the residuals from the two test problems. Our implementation can be downloaded at

`http://seesar.lbl.gov/anag/staff/wen/download.html`.

References

[1] Ann Almgren, Thomas Buttke, and Phillip Colella. A fast adaptive vortex method in three dimensions. *Journal of computational Physics*, 113(2):177-200, 1994.

[2] Ann Almgren, John Bell, Phillip Colella, Louis Howell, and Michael Welcome. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. *Journal of computational Physics*, 42:1-46, 1998.

[3] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994.

[4] John Bell, Marsha Berger, Jeff Saltzman, and Mike Welcome. Three dimensional adaptive mesh refinement for hyperbolic conservation laws. *Journal of Scientific Computing*, 15(1):127-138, 1994.

[5] J. B. Bell. AMR for low Mach number reacting flows. LBNL Report LBNL-54351, Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, 2003.

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

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

[8] M. J. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions Systems, Man and Cybernetics*, 21(5):1278-1286, 1991.

[9] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial, Second Edition*. SIAM, Philadelphia, PA, 2000.

[10] J. Cary, S. Shasharina, J. Cummings, J. Reynders, and P. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. Report No. LA-UR-96-4064, Los Alamos National Laboratory, 1996.

[11] CCSE Applications Suite website: `http://seesar.lbl.gov/CCSE/Software/index.html`. Center for Computational Sciences and Engineering (CCSE), Lawrence Berkeley National Laboratory, Berkeley, CA.

[12] Chombo website: `http://seesar.lbl.gov/ANAG/software.html`. Applied Numerical Algorithms Group (ANAG), Lawrence Berkeley National Laboratory, Berkeley, CA.

[13] P. Colella, M. Dorr, and D. Wake. Numerical solution of plasma-fluid equations using locally refined grids. *Journal of computational Physics*, 152:550–583, 1999.

[14] W. Y. Crutchfield and M. Welcome. Object-oriented implementation of adaptive mesh refinement algorithms. *Scientific Programming*, 2(4):145–156, 1993.

[15] A. L. Garcia, J. B. Bell, W. Y. Crutchfield, B. J. Alder. Adaptive mesh and algorithm refinement. *Journal of computational Physics*, 154:134-155, 1999.

[16] R. Hornung and J. A. Trangenstein. Adaptive mesh refinement and multilevel iteration for flow in porous media. *Journal of computational Physics*, 136(2):522–545, 1997.

[17] L. H. Howell and J. B. Bell. An adaptive-mesh projection method for viscous incompressible flow. *SIAM Journal of Scientific Computing*, 18(4):996-1013, 1997.

[18] L. H. Howell, R. B. Pember, P. Colella, J. P. Jessee, and W. A. Fiveland. A conservative adaptive-mesh algorithm for unsteady, combined-mode heat transfer using the discrete ordinates method. *Numerical Heat Transfer, Part B: Fundamentals*, 35:407–430, 1999.

[19] J. P. Jessee, W. A. Fiveland, L. H. Howell, P. Colella, and R. B. Pember. An adaptive mesh refinement algorithm for the radiative transport equation. *Journal of computational Physics*, 139(2):380–398, 1998.

[20] R. I. Klein, J. B. Bell, R. B. Pember, T. Kelleher. Three dimensional hydrodynamic calculations with adaptive mesh refinement of the evolution of Rayleigh Taylor and Richtmyer Meshkov instabilities in converging geometry: multimode perturbations. *Proceedings of the 4th International Workshop on Physics of Compressible Turbulent Mixing*, Cambridge, England, March 1993.

[21] D. F. Martin and K. L. Cartwright. Solving Poisson's equation using adaptive mesh refinement. Technical Report UCB/ERI M96/66, UC Berkeley, 1996.

[22] D. Martin and P. Colella. A cell-centered adaptive projection method for the incompressible Euler equations. *Journal of computational Physics*, 163:271-312, 2000.

[23] D. F. Martin. Draft: Adaptive Mesh Refinement for incompressible Navier-Stokes equations. Applied Numerical Algorithms Group (ANAG), Lawrence Berkeley National Laboratory, Berkeley, CA.

[24] Matthew Tyler Bettencourt. *A Block-Structured Adaptive Steady–State Solver for the Drift–Diffusion Equations*. PhD thesis, Dept. of Mechanical Engineering, Univ. of California, Berkeley, 1998.

[25] D. L. Modiano and E. M. Murman. Adaptive computations of flow around a delta wing with vortex breakdown. *AIAA Journal*, 32(7):1545-1547, 1994.

[26] R. B. Pember, J. B. Bell, P. Colella, W. Y. Crutchfield, and M. L. Welcome. An adaptive Cartesian grid method for unsteady compressible flow in irregular regions. *Journal of computational Physics*, 120:278–304, 1995.

[27] R. B. Pember, L. H. Howell, J. B. Bell, P. Colella, W. Y. Crutchfield, W. A. Fiveland, and J. P. Jessee. An adaptive projection method for unsteady, low Mach number combustion. *Combustion Science and Technology*, 140:123–168, 1998.

[28] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Crutchfield, and John B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms. *Computing and Visualization in Science*, 3:147-157, 2000.

[29] E. Steinthorsson, D. Modiano, P. Colella. Computations of unsteady viscous compressible flows using adaptive mesh refinement in curvilinear body-fitted grid systems. NASA technical memorandum 106704, ICOMP report no. 94-17, 1994.

[30] M. C. Thompson and J. H. Ferziger. An adaptive multigrid technique for the incompressible Navier-Stokes equations. *Journal of computational Physics*, 82:94-121, 1989.

[31] Titanium website: `http://www.cs.berkeley.edu/projects/titanium/`. Titanium Group, Dept. of Computer Science, University of California, Berkeley, CA.

[32] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken. Titanium: a high-Performance Java dialect. *ACM 1998 workshop on Java for high-performance computing*, Stanford, CA, 1998.