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] ...). 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 $\Delta x$ is the mesh size, $M(\cdot)$ is a $(p + q)$-order differential operator, and $\phi$ is the solution. If $\phi$ is not smooth, then a small $\Delta x$ 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 $l_{\text{max}}$ (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

\[ L\phi = f \text{ in } \Omega, \quad (2.1) \]

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

\[ \phi = g \text{ on } \partial \Omega. \quad (2.2) \]

As a matter of notation, for each refinement level \( l \), we denote by \( \Omega_l \) the union of rectangular grids at this level. Let \( \phi^l \) and \( f^l \) be the discretized versions of \( \phi \) and \( f \) defined on \( \Omega_l \). 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 \( \phi^l \) involves not only \( \phi^l \) itself but also \( \phi^{l-1} \) and \( \phi^{l+1} \) (assuming \( 0 < l < l_{\text{max}} \)).

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 RectDomains, 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, \( \phi^l \) 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...
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 \textit{copy} method of Titanium array. Given two Titanium arrays $T.A_1$ and $T.A_2$, $T.A_1.copy(T.A_2)$ copies the contents of the elements of $T.A_2$ into the elements of $T.A_1$ that have the same indexes, where $T.A_1$ and $T.A_2$ can belong to different processors. For our Poisson solver, a large portion of the communication time is consumed by the \textit{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 \textit{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 $\phi^l$ for $l = 0, \ldots, l_{\text{max}}$ 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 $\Omega^0, \ldots, \Omega^{l_{\text{max}}}$. Hereafter, we denote this composite solution by $\phi^{\text{comp}}$, and let $L^{\text{comp}}$ be the corresponding discretized operator.

At each level $l$, on the inner regions of $\Omega^l$ 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 $\Omega^l$. In the multigrid algorithm described here, a simplified version of $L^l$ denoted by $L^l_{\text{af}}$ 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():
\[ R^{\text{comp}} := f^{\text{comp}} - L^{\text{comp}}(\phi^{\text{comp}}) \]
while \([\|R^{\text{comp}}\| > \epsilon\|f^{\text{comp}}\|]\)
\[ AMRVCycle(l_{\text{max}}): \]
\[ R^{\text{comp}} := f^{\text{comp}} - L^{\text{comp}}(\phi^{\text{comp}}) \]
end while

procedure AMRVCycle(level l):
if \((l = l_{\text{max}})\) then \(R^l := f^l - L_{\text{init}}(\phi^l, \phi^{l-1})\)
if \((l > 0)\) then
\[ \phi^{l}_{\text{copy}} := \phi^l \text{ on } \Omega^l \]
\[ e^l := 0 \text{ on } \Omega^l \]
mgRelax(e^l, R^l, r^l)
\[ \phi^l := \phi^l + e^l \]
\[ e^{l-1} := 0 \text{ on } \Omega^{l-1} \]
\[ R^{l-1} := \text{Average}(R^l - L_{\text{init}}(e^l, e^{l-1})) \text{ on } \Omega^{l-1} \]
\[ R^{l-1} := f^{l-1} - L^{l-1}(\phi^{\text{comp}}) \text{ on } \Omega^{l-1} \]
AMRVCycle(l - 1)
\[ e^l := e^l + \text{Interpolate}(e^{l-1}) \]
\[ R^l := R^l - L_{\text{init}}(e^l, e^{l-1}) \]
mgRelax(\(\delta e^l\), \(R^l\), \(r^l\))
\[ e^l := e^l + \delta e^l \]
\[ \phi^l := \phi^{l}_{\text{copy}} + e^l \]
else
solve \(L^{0}_{\text{init}}(e^0) = R^0 \text{ on } \Omega^0\)
\[ \phi^0 := \phi^0 + e^0 \]
end if

procedure GSRB(\(\varphi^l\), \(R^l\))
\{  
for \(i = 1, \ldots, \text{NumSmoothDown}\)
\hspace{1em} GSRB(\(\varphi^l\), \(R^l\))
end for
if \((r > 2)\) then
\[ \delta^r := 0 \]
\[ R^r := \text{Average}(R^l - L^l(\varphi^l, \varphi^r \equiv 0)) \]
mgRelax(\(\delta^r\), \(R^r\), \(r/2\))
\[ \varphi^l := \varphi^l + \text{Interpolate}(\delta^r) \]
for \(i = 1, \ldots, \text{NumSmoothUp}\)
\hspace{1em} GSRB(\(\varphi^l\), \(R^l\))
end if
\}

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

procedure mgRelax(\(\varphi^l\), \(R^l\), \(r\))
\{  
for \(i = 1, \ldots, \text{NumSmoothDown}\)
\hspace{1em} GSRB(\(\varphi^l\), \(R^l\))
end for
if \((r > 2)\) then
\[ \delta^r := 0 \]
\[ R^r := \text{Average}(R^l - L^l(\varphi^l, \varphi^r \equiv 0)) \]
mgRelax(\(\delta^r\), \(R^r\), \(r/2\))
\[ \varphi^l := \varphi^l + \text{Interpolate}(\delta^r) \]
for \(i = 1, \ldots, \text{NumSmoothUp}\)
\hspace{1em} GSRB(\(\varphi^l\), \(R^l\))
end if
\}

Figure 4. Gauss-Seidel relaxation with red-black ordering. Here \(\lambda\) is the relaxation parameter.

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

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
\[
\|R^{\text{comp}}\|_{\infty} \leq \text{10}^{-10}\|R^{\text{init}}\|_{\infty}.
\]

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
\[
L_{\text{init}}(e^0) = R^0 \text{ on } \Omega^0,
\]
which however is not performance critical. Therefore, we do not expect that the residual norms from both sides match exactly.
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/users/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 slower.
Titanium is a high-level parallel programming 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-
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

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

<table>
<thead>
<tr>
<th>iteration</th>
<th>Titanium</th>
<th>Chombo</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>6.144</td>
<td>6.144</td>
</tr>
<tr>
<td>1</td>
<td>0.273</td>
<td>0.273</td>
</tr>
<tr>
<td>2</td>
<td>0.254</td>
<td>0.254</td>
</tr>
<tr>
<td>3</td>
<td>0.009</td>
<td>0.009</td>
</tr>
<tr>
<td>4</td>
<td>3.780E-04</td>
<td>3.800E-04</td>
</tr>
<tr>
<td>5</td>
<td>5.093E-06</td>
<td>4.748E-06</td>
</tr>
<tr>
<td>6</td>
<td>2.090E-07</td>
<td>1.570E-07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>iteration</th>
<th>Titanium</th>
<th>Chombo</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>9.830E+04</td>
<td>9.830E+04</td>
</tr>
<tr>
<td>1</td>
<td>4.169</td>
<td>4.169</td>
</tr>
<tr>
<td>2</td>
<td>1.290</td>
<td>1.277</td>
</tr>
<tr>
<td>3</td>
<td>0.0222</td>
<td>0.0219</td>
</tr>
<tr>
<td>4</td>
<td>1.039E-03</td>
<td>1.039E-03</td>
</tr>
<tr>
<td>5</td>
<td>2.128E-05</td>
<td>2.128E-05</td>
</tr>
<tr>
<td>6</td>
<td>7.367E-07</td>
<td>7.367E-07</td>
</tr>
</tbody>
</table>

References


