An Adaptive Mesh Refinement Benchmark for Modern Parallel Programming Languages

Tong Wen
IBM T. J. Watson Research Center
Hawthorne, NY 10532
tongwen@us.ibm.com

Jimmy Su
University of California, Berkeley
Berkeley, CA 94720
jimmysu@eecs.berkeley.edu

Phillip Colella
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
colella@hpcrd.lbl.gov

Katherine Yelick
University of California, Berkeley and Lawrence Berkeley National Laboratory
Berkeley, CA 94720
yelick@eecs.berkeley.edu

Noel Keen
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
noel@hpcrd.lbl.gov

ABSTRACT

We present an Adaptive Mesh Refinement benchmark for evaluating programmability and performance of modern parallel programming languages. Benchmarks employed today by language developing teams, originally designed for performance evaluation of computer architectures, do not fully capture the complexity of state-of-the-art computational software systems running on today's parallel machines or to be run on the emerging ones from the multi-cores to the peta-scale High Productivity Computer Systems. This benchmark extracted from a real application framework presents challenges for a programming language in both expressiveness and performance. It consists of an infrastructure for finite difference calculations on block-structured adaptive meshes and a solver for elliptic Partial Differential Equations built on this infrastructure. Adaptive Mesh Refinement algorithms are challenging to implement due to the irregularity introduced by local mesh refinement. We describe those challenges posed by this benchmark through two reference implementations (C++/Fortran/MPI and Titanium) and in the context of three programming models.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; G.4 [**Mathematics of Computing**]: Mathematical Software—*parallel and vector implementations*; J.2 [**Computer Applications**]: Physical Sciences and Engineering

Keywords

Adaptive mesh refinement, Benchmark, Parallel programming languages, Performance, Programmability, Scalability

1. INTRODUCTION

There is a constant tension in parallel programming model efforts between ease-of-use and high performance, and the game is to strike the right balance between these two competing goals. As an explosive increase in hardware parallelism is expected in the near future, new programming paradigms intended for delivering high productivity on such emerging computer architectures are of great research interest. Productivity becomes the measure of success because past experience has shown that programming high quality software on machines that embody a rich set of hardware parallelism is difficult and would be very costly without the right language and tool support. The community obviously needs adequate benchmarks to study the productivity of a programming language system (language design, compiler, runtime, and tool infrastructure) and longs for convincing show cases to promote such a system. Benchmarks employed today by language developing teams such as the Numerical Aerodynamic Simulation Parallel Benchmarks (NPB) [1] were originally designed for performance evaluation of computer architectures. They are simplified numerical kernels and do not fully capture the complexity of state-of-the-art computational software systems running on today's parallel machines or to be run on the emerging ones from the homogeneous multi-cores, to the heterogeneous accelerators, and to the High Productivity Computer Systems (HPCS) [11] designed to deliver peta-scale performance. By extracting key components from a real application framework, we have devised an Adaptive Mesh Refinement (AMR) benchmark for evaluating a parallel programming language's programmability and performance. Implementing this benchmark with good performance and scalability is considered as a highly challenging problem.

In this paper we describe the challenges posed by this benchmark in both expressiveness and performance through two reference implementations: one written in C++ and Fortran 77 with MPI and the other in Titanium – an explicitly parallel dialect of Java designed for high-performance scientific computing [21, 20, 19]. We also discuss how to address these challenges in the context of three programming models: Single Program Multiple Data (SPMD) with twosided communication (MPI), SPMD with Partitioned Global Address Space (PGAS) and one-sided communication (Titanium), and Task Parallelism with PGAS and one-sided communication (X10). This project started with porting Chombo [5] to Titanium in order to explore AMR algorithms for ocean modeling. Chombo is a widely used library written in C++ and Fortran 77 with MPI for blocked-structured AMR finite difference discretization. We then found that the component for solving elliptic Partial Differential Equations (PDE) in Chombo's framework would serve as a good benchmark for evaluating programmability and performance of modern parallel programming languages, for examples, the PGAS languages: Co-Array Fortran, Unified Parallel C (UPC) and Titanium, as well as the recent HPCS languages: Chapel by Cray Inc., Fortress by Sun Microsystems and X10 by IBM.

Due to the irregularity introduced by local mesh refinement, AMR algorithms are not easy to program particularly with good performance and scalability (considering the high surface-volume ratio of AMR grids). In addition to regular operations one may find in ordinary finite difference calculations, this class of applications typically involves irregular (hierarchical, pointer-based) data structures, input-dependent computational load which easily requires domain-specific and dynamic load balancing, as well as fine-grained communication and irregular operations for updating grid boundaries in the adaptive mesh hierarchy. Chombo uses object orientation with modular and layered design to increase modularity and reusability of its codes, and to hide details at different levels behind interfaces. Each solver component in Chombo is built on a layer of AMR data structures and a layer of commonly used AMR operations. The corresponding Titanium implementation of this benchmark follows Chombo's design which includes:

1. A fixed-size grid generator (different from its Chombo counterpart)
2. A load balancer using Space-Filling Curves (SFCs) (different from its Chombo counterpart)
3. An infrastructure for AMR applications comprised of AMR data structures and basic AMR operations
4. An elliptic PDE solver
5. Test problems and examples of application code

Incorporating the layers of AMR data structures and basic AMR operations into this benchmark reflects the software structure of Chombo and in general how the library approach would address the programming difficulties incurred by local mesh refinement. Unlike others, this benchmark imposes constraints on the quality of its implementation. In this context, better software quality means higher composability, reusability and maintainability. The reason to choose the elliptic PDE solver is due to its wide usage as a numerical kernel in various problems and its relative simplicity compared to Chombo's time-dependent components in that dynamic regridding and load balancing is not required in this solver.

The rest of this paper is organized as follows. Section 2 provides a brief introduction to AMR, Chombo, as well as the Titanium and X10 programming languages so as to make this paper self-contained. Section 3 defines the test problems where a scalar 3D Poisson's equation is solved subject to Dirichlet boundary condition. Section 4 describes the high-level abstractions involved in this benchmark and the issues in expressing them. Section 5 discusses the performance and scalability aspects of this benchmark. The key factors to consider in mapping a programming model onto a parallel computer architecture are the granularity of parallelism, the cost of data transfer and the efficiency of synchronization. In this section, we discuss issues such as load balancing, communication pattern and optimization, and utilization of fine-grained parallelism, for example, how to overlap communication with computation. An interesting question to consider here is whether we can scale AMR up to tens or hundreds of thousands processing elements. Finally, in section 6 we draw our conclusions and list future work.

In this paper, we assume that only one process is assigned to a processor so that they can be used interchangeably.

2. AMR, CHOMBO, TITANIUM, AND X10

2.1 Block-structured AMR

The AMR methodology has been successfully applied to numerical modeling of various physical problems that exhibit multiscale behavior, such as those mentioned in [9]. The idea of AMR is quite straightforward, that is, to apply finer discretization only at places where higher resolution is needed. However, the simplicity of finite difference calculation on a uniform grid is traded in AMR, where the irregularity comes from the boundaries between grids introduced by local mesh refinement.

In our AMR implementation, numerical solutions are defined on a hierarchy of nested rectangular grids. The grids from the same level of discretization, that is, with the same grid spacing (cell size), are organized together and each level of grids are embedded in the next coarser level recursively subject to certain nesting conditions [9]. Figure 1 depicts a portion of a grid hierarchy in two dimensions. In AMR applications there are usually multiple levels of refinement, but only two adjacent levels are shown here. Note that the underlying problem domain is not shown in this figure which is discretized using one rectangular grid at each refinement level. In this example, three fine grid patches are superimposed on two coarse ones. Each solid box shown here represents a rectangular grid, while each dotted square represents a grid cell. Thus, an array defined on the third fine grid, for example, will have 16 elements indexed by the location of each cell. Grids are often accreted to accommodate ghost cells for handling various boundary conditions. For example, to apply a five-point stencil operation on fine grid 0, values from fine grid 1 are needed along the boundary between these two grids, which are cached as ghost values. For ghost cells not covered by fine grid 1, the corresponding ghost values have to be interpolated using information from fine grid 0 and from coarse grids 0 and 1. Also, to apply this stencil operation on fine grid 2, part of the ghost values have to be interpolated using the physical boundary condition. The nesting conditions we use are more general than others in that a fine grid is allowed to cover multiple ones at the next coarser level and grids of different sizes (in terms of the number of cells a grid contains) may be used at each level. In this benchmark, grid configuration is static. However, in other applications it may need to change from time to time to adaptively match the evolution of numerical

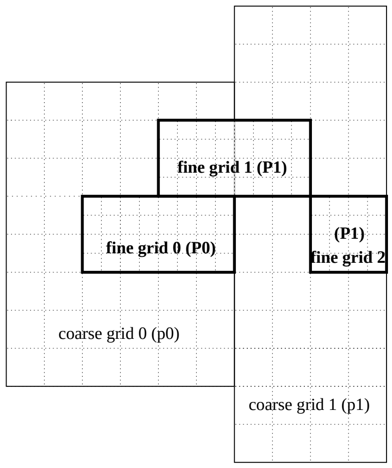


Figure 1: Two adjacent levels of grid patches in a grid hierarchy are shown in this example for the two-dimensional case. Each solid box represents a rectangular grid, and the dotted squares inside a box are the cells it contains. There are two grids at the coarse level and three at the fine level. For each grid, the number in parenthesis represents the process it is assigned to. The refinement ratio between these two refinement levels is 2 in each dimension. Note that the underlying problem domain is not shown in this figure which is discretized using one rectangular grid at each refinement level.

solution, resulting in dynamic regridding and load balancing.

In summary, three kinds of basic operations are involved in AMR algorithms:

1. Regular (local) stencil operations on each grid
2. Copying values from one grid to another such as in exchanging ghost values between grids at the same level
3. Irregular computations for updating boundary values, for example, the location-dependent quadratic interpolation of ghost values at the coarse-fine grid interface [9].

AMR algorithms are challenging to implement not only because of the irregular computations and data access on grid boundaries, but also because of the complexity in which the interactions between levels of refinement are orchestrated. We do not cover the details of the algorithms implemented in this benchmark, which are well documented in [17, 14, 9]. Here, all algorithms are cell-centered, and the refinement ratio is uniform in each spatial dimension. Development of a highly scalable Poisson solver is considered as one of the most challenging goals for AMR.

2.2 Chombo Library

Chombo developed at Lawrence Berkeley National Lab (LBNL) is a library for block-structured AMR applications. It is written in C++ and Fortran 77 with MPI and designed to ameliorate the programming difficulties posed by AMR algorithms particularly for traditional Fortran programmers to whom AMR data structures are unfamiliar. Chombo utilizes C++'s expressiveness for the implementation of complex data structures and irregular operations, while passes arrays to Fortran subroutines for high-performance bulk rectangular operations. It also augments the C++ and Fortran interface with a macro package to shorten argument lists and declarations, to enable dimension-independent Fortran programming, and to make compiler-and-platform independent the calling of Fortran subroutines from C++. Most of the Chombo classes are templated for generality, and inheritance is used to define interfaces.

Chombo uses object orientation with modular and layered design to increase modularity and reusability of its codes, and to hide details at different levels behind interfaces. Each solver component in Chombo is built on a library of AMR data structures and a library of commonly used AMR operations. The Chombo framework at the present time consists of five components. The *BoxTools* library implements high-level abstractions of AMR data structures along with associated calculus. The data communication layer using MPI is also implemented in this library and hidden from application users. The *AMRTools* library consists of classes which implement a number of operations that often appear in AMR algorithms, supporting extensions of second-order accurate discretizations of quasi-linear elliptic, parabolic, and hyperbolic PDEs in conservation form to AMR. The other three are solver components: *AMRTimeDependent*, *AMRElliptic*, and *ParticleTools*. Each of them support a particular class of applications along with examples of its usage. AMRElliptic is the one we have implemented in Titanium which consists of classes supporting an AMR-Multigrid algorithm for elliptic PDEs. Besides these components, Chombo also supports I/O based on HDF5 [12] and has a visualization library called ChomboVis.

The parallel model of Chombo is SPMD with bulk synchronous communication, where communication and computation are performed in different phases separated by global barriers. In Chombo, MPI communications are transparent to users and programmed in a low-level style so as to achieve good performance.

2.3 Titanium: an Explicitly Parallel Dialect of Java

The Titanium language is an explicitly parallel dialect of Java designed for high-performance scientific programming. Its model of parallelism is static SPMD augmented with a shared-memory like abstraction called Partitioned Global Address Space. Compared to conventional approaches to parallel programming where standard sequential programming languages (Fortran and C/C++) are extended with a message passing library, we argue that Titanium allows more concise and user-friendly syntax, and makes more information explicitly available to the compiler. Its goal is to achieve greater expressive power without sacrificing parallel performance.

Titanium extends sequential Java with the following key features. Most of them will be covered during our later discussion.

1. **Titanium arrays** – Titanium provides a powerful multidimensional array abstraction defined on a global index space along with the same kinds of sub-array operations available in Fortran 90.

2. **Domain calculus** – The built-in multidimensional domain calculus provides syntactical support for sub-arrays. In Titanium, the location of a cell as shown in Figure 1 is represented by an integer vector called *point*, and a *domain* is a set of points which can be either rectangular or not. Points (`Point`) and domains (`RectDomain, Domain`) are first-class types and literals.

3. **Foreach loops** – The iteration over any multidimensional Titanium array is expressed concisely in one loop, using the unordered looping construct `foreach`.

4. **Distributed data structures** – In Titanium, the parallelism comes from the distribution of data across processes. It is each process's responsibility to construct its local share of a global data structure and have the references available to others. After that each process can access the entire data structure in the same way as on a shared-memory machine. Global data structures are built in Titanium through its general pointer-based distribution mechanism. The bulk communication between two processes is realized through the `copy` method of Titanium arrays.

5. **Templates** – Titanium supports templates similar to those in C++, allowing us to mirror the templated Chombo interface.

6. **Non-blocking array copy** – The support of non-blocking array copy enables the overlap of computations and communications.

7. **The local keyword and locality qualification** – Titanium expresses the affinity between data and its owning process explicitly using the `local` type qualifier. Locality information is automatically propagated by the Titanium optimizer using a constraint-based inference. This performance feature helps especially when running computations on distributed-memory platforms.

8. **Immutable classes and operator overloading** – Application-specific primitive types can be defined in Titanium as immutable classes. Objects of an immutable class (lightweight objects) are unboxed, analogous to C structs. They are manipulated and passed by value.

9. **Memory management using Regions** – In addition to its garbage collector, Titanium allows programmers to explicitly release memory allocations through its zone-based memory management.

10. **Cross-language calls** – Titanium code can call libraries written in other languages, such as FFTW [10].

For more details of Titanium, readers please refer to [8]. The Titanium implementation of AMR covers most of the above features.

2.4 X10: Designed for High Productivity

Although we already have two reference implementations, it would be interesting to look at this benchmark from the perspective of another language. X10 is a new PGAS language being developed at IBM as part of the DARPA HPCS project [4], which is also based on sequential Java with extensions for large scale and heterogeneous parallel programming.

The fundamental distinction between X10 and other PGAS languages is that its model of parallelism is Task Parallelism. X10 supports the notion of *place* – a place may be considered as a virtual shared-memory multiprocessor (SMP). In X10, *activities* (lightweight threads) and data structures have affinity with a place; activities can be spawned recursively at any place either locally or remotely; users have full control over the distribution of data structures across places and can define their own distributions. Besides constructs for converting the local termination of an activity to global termination, X10 uses *clock* to coordinate activities and supports unconditional and conditional atomic blocks. The array abstractions in X10 are similar to those in Titanium. There are other interesting features of X10 such as annotations and dependent types which we do not have space to cover here, readers please refer to X10's website [18] for more details and examples.

3. TEST PROBLEMS

The test problem is a scalar 3D Poisson's equation

$$L\phi = f \text{ in } \Omega, \tag{1}$$

subject to Dirichlet boundary condition

$$\phi = g \text{ on } \partial\Omega. \tag{2}$$

Here, L is the Laplacian operator and Ω is an open cubic domain of unit size. The lower left corner of Ω is at the origin of the coordinate system. The right-hand side f will be defined shortly.

Two setups of the above problem are used: one for small runs and the other for large runs. The small test problem has two vortex rings in its problem domain, while the large one has single ring. The parameters of these vortex rings are listed in Table 1.

The grid configurations in our test problems are static, that is, once they are generated they do not change during the computation. The right-hand side f and the tagging criteria for AMR grid construction are defined in the following.

For each vortex ring centered at (X, Y, Z), a quantity ω at position (x, y, z) is computed as follows:

$$(x, y, x) = (x, y, z) - (X, Y, Z) \tag{3}$$

$$r = \sqrt{x^2 + y^2} \tag{4}$$

$$\sin\theta = \frac{y}{r} \text{ and } \cos\theta = \frac{x}{r} \tag{5}$$

$$x_0 = R\cos\theta \text{ and } y_0 = R\sin\theta \tag{6}$$

$$d = \sqrt{(x - x_0)^2 + (y - y_0)^2 + z^2} \tag{7}$$

$$\omega = \frac{\omega_0}{\sigma^2\alpha} e^{(-d/\sigma)^3}. \tag{8}$$

If there are multiple vortex rings, ω is just the sum of the contributions from each one. The right-hand side f for the large test problem is defined as

$$f = -\omega\sin\theta, \tag{9}$$

and for the small test problem, it is defined as

$$f = -(\omega_1\sin\theta_1 + \omega_2\sin\theta_2), \tag{10}$$

Configuration	Parameter			
	Center (X, Y, Z)	Radius R	Thickness σ	Strength ω_0
Small	$(0.5, 0.5, 0.4)$	0.2	0.0275	1.5
	$(0.5, 0.5, 0.65)$	0.25	0.0275	1.0
Large	$(0.5, 0.5, 0.4)$	0.2	0.0275	1.5

Table 1: Parameters for the vortex rings in the test problems. For all the vortex rings, the scaling factor α is 2268.85.

where ω_i and θ_i are the quantities corresponding to the ith vortex ring. For simplicity, g is set to zero in the boundary condition. The tagging criteria for AMR grid construction is

$$\omega > 0.16. \tag{11}$$

That is, when this condition is true, local refinement is required. The convergence criteria is

$$||\rho_m||_\infty \leq 10^{-10}||\rho_0||_\infty, \tag{12}$$

where ρ_i is the residual on the AMR grids at step i.

Grid configurations along with process assignments can be directly imported from Chombo, whose grid generator uses grids of variable sizes [3]. Figure 2 shows the small and large grid configurations generated by Chombo. People are allowed to use their own grid generators. We use a simple fixed-size grid generator for the Titanium implementation, but our experience shows that there is no performance difference between these two grid generators for the test problems.

4. HIGH-LEVEL ABSTRACTIONS AND DEVELOPMENT PRODUCTIVITY

In this section, we describe the high-level abstractions involved in block-structured AMR. Providing high-level abstractions is a key way to ease the developing efforts of application programmers, but greatly subject to the performance constraint. C++ and Titanium support similar object orientation and generic programming, enabling us to mirror Chombo's design of software architecture without significant modifications. However, the major difference is that most of the abstractions introduced below are supported at the language level in Titanium. Examples to be shown in the following are all written in Titanium for the purposes of conciseness and generality. Please also note that X10 supports abstractions similar to those described below.

4.1 Points, Domains, Rectangular Arrays, and Data Holders

Titanium arrays are built on rectangular domains and indexed by points. Remember the definition from the previous section: a point is an integer vector and a domain is a set of points. For example, fine grid 0 in Figure 1 can be expressed as

```
Point<2> low = [0,0], high = [7,3];
RectDomain<2> fineGrid0 = [low:high];
```

or simply as

```
RectDomain<2> fineGrid0 = [[0,0]:[7,3]];
```

assuming its lower left corner cell is the origin. In AMR, grids are often accreted to accommodate a surrounding layer of ghost cells for handling various boundary conditions. To add a ghost layer of depth one to `fineGrid0` we can either redefine it by specifying the lower left and the upper right corners or by calling the `accrete` method as in the following statement.

```
fineGrid0 = fineGrid0.accrete(1);
```

Points and domains are first-class entities in Titanium – they can be stored in data structures, specified as literals, passed as values to methods and manipulated using their own set of operations.

The following statement defines a double array on `fineGrid0`.

```
double [2d] fineArray0 = new double [fineGrid0];
```

Note that the current value of `fineGrid0` is `[[-1,-1]:[8,4]]`. Titanium arrays are defined on a global index space where they may start at any base point. On machines with distributed memory systems, `fineArray0` resides in memory with affinity to exactly one process, namely the process that executes the above statement. However, it is accessible globally as it were on a shared-memory machine.

Titanium supports sub-arrays by creating views (data aliasing) of an array's data without incurring any copy of the data. For example, to create a view of `fineArray0` excluding its ghost elements, we can use the statement

```
double [2d] fineArray0Interior =
                        fineArray0[[0,0]:[7,3]];
```

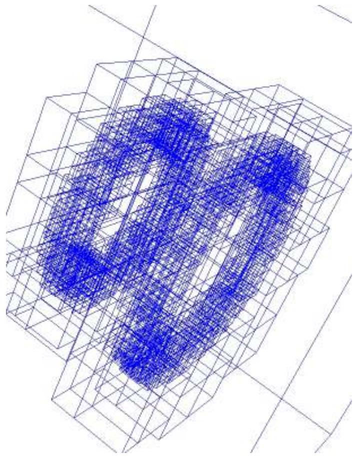
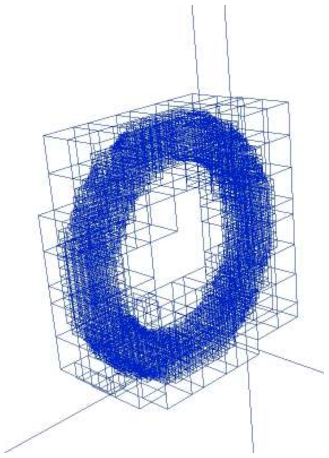
or use the `shrink` method

```
double [2d] fineArray0Interior =
                        fineArray0.shrink(1);.
```

Bulk data transfers in Titanium are realized through array copies. The following example shows how two arrays logically adjacent to each other exchange values using the `copy` method and cache them in ghost regions.

```
// array defined on fine grid 1 with ghost region
double [2d] fineArray1 = new double [[3,3]:[12,8]];
 ...
double [2d] fineArray1Interior =
                        fineArray1.shrink(1);
// fineArray0 is the destination, and
// fineArray1Interior is the source
fineArray0.copy(fineArray1Interior);
finaArray1.copy(fineArray0Interior);
```

These two arrays may belong to different processes and may live in different nodes of a cluster. To copy arrays whose domains are disjoint, Titanium provides the `translate` method to shift the indices of an array view by logically adding a given point (as an argument to this method) to every index in this array view, so that the new view overlaps with the array to be copied to or from. Built-in methods such as `copy`

Small configuration			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 2: The small and large grid configurations from Chombo. Only the top and middle levels are shown in this figure. Each box here represents a three-dimensional grid patch. At the base level, the union of the grid patches fully covers the problem domain. There are totally 32^3 and 128^3 grid cells at the base levels of the small and large configurations respectively. The refinement ratio between any two adjacent levels is 4 in each spatial dimension for both cases.

are not opaque to the Titanium compiler, which recognizes and treats such method calls specially and applies optimizations to them, for example, turning blocking operations into non-blocking ones.

Iterating through a multidimensional array is concisely expressed in Titanium using an unordered looping construct `foreach`. The following example shows how a 5-point stencil operation is implemented.

```
dobule c0, c1;
 ...
final Point<2> EAST = [1,0];
final Point<2> WEST = [-1,0];
final Point<2> NORTH = [0,1];
final Point<2> SOUTH = [0,-1];
final RectDomain<2> interior =
                    fineArray0.domain().shrink(1);
double [2d] result = new double [interior];
foreach (p in interior){
    result[p]= c0 * fineArray0[p] +
               c1 * ( fineArray0[p + EAST] +
                      fineArray0[p + WEST] +
                      fineArray0[p + NORTH] +
                      fineArray0[p + SOUTH] );
}
```

In this example, for the ghost values not covered by fine grid 1 they are updated with certain interpolation procedure that may involve data from the coarse level and become location dependent. In other languages, iteration over a multidimensional array would require a multi-level loop nest. If the order of iteration is irrelevant to a computation, using a `foreach` loop allows the compiler to reorder loop iterations to maximize performance, for example, by performing automatic cache blocking and tiling optimizations. It also simplifies optimizations such as bound checking elimination and array access strength reduction.

In C++, abstractions such as points, domains, and multidimensional rectangular arrays have to be implemented as user-defined classes. In Chombo, a rectangular array defined on a n-dimensional domain is a $n+1$-dimensional array. That is, there is a one-dimensional array containing m state variables for each point in this domain (vector field). A more general representation of Chombo's rectangular array in Titanium is an array whose elements are of an immutable type that encapsulates the state variables. Immutable classes are application specific primitive types whose objects are lightweight (unboxed objects analogous to C structs). In Titanium, immutables are manipulated and passed by value, avoiding pointer-chasing overheads that would otherwise be associated with the use of tiny objects in Java. A good example of immutable class would be the `Complex` class used in the Titanium implementation of NPB Fourier Transform benchmark [7].

```
public immutable class Complex{
    public double real;
    public double imag;
    public inline Complex(double r, double i){
        real = r; imag = i;
    }
    // operater overloading
    public iline Complex op+(Complex c) {
        return new Complex(c.real + real,
            c.imag + imag);
    }
    ...
}
```

Immutable types such as Complex are not subclasses of java.lang.Object and induce no overheads for pointers or object headers. They are implicitly final, that is, they never pay execution-time overheads for dynamic method call dispatch. More importantly to us, an array of Complex immutables is stored in a single contiguous piece of memory consisting of all the real and imaginary parts. In the Titanium implementation of AMR, a general rectangular array is represented as a templated wrapper class BoxedArray<T> which contains a Titanium array of type T. An instantiation of this wrapper class, for example, can be BoxedArray<double> or BoxedArray<Complex>. Titanium supports C++ like generic types. In contrast to Java, Titanium allows values and primitive types as template parameters. Note that Titanium's support for generic types long predates that of Java (version 5.0 in August 2004).

A level of domains and a level worth of data which resides on these domains are contained in user-defined classes. The meta-data class BoxLayout contains an array of domains at a refinement level along with their process assignments, as well as methods that operate on them such as coarsen and refine. The data-holder class BoxLayoutData<T> contains the data defined on the meta-data class, which is templated for generality. The template parameter T defines the data type on each domain, for example, in our Titanium implementation it can be BoxedArray<Complex>. In Titanium, templated classes can be used as generic parameters. The data-holder class has two communication methods: exchange and copy. The first method exchanges values between neighboring BoxedArray<Complex>s, for example, whose domains are adjacent but disjoint, and then cache the values in local ghost regions. The second one copies the data of a data-holder to that of another one at the intersections of their domains. Compared with MPI, Titanium's array copy method as a way to realize one-sided bulk communications makes the implementation of these two methods very straightforward and expressed at a high-level abstraction of logic. We will address the related performance issues in the next section.

4.2 Development Productivity

Chombo uses C++ to implement the above high-level abstractions, while having Fortran to handle regular array operations for high performance. At the time when Chombo was designed, there was in fact no viable alternative to MPI for people to consider, which has been the de facto standard. Mixed-language programming turns out to be difficult for debugging and maintaining of a system like Chombo. Although Chombo provides a macro package to ease the writing of Fortran subroutines called from C++, it still can not match the conciseness Titanium provides. The following is a Chombo Fortran subroutine computing the dot product of two Chombo rectangular arrays.

```
c words with CHF prefix are Chombo Fotran macros.
subroutine DOTPRODUCT(
     &     CHF_REAL[dotprodout],
     &     CHF_CONST_FRA[afab],
     &     CHF_CONST_FRA[bfab],
     &     CHF_BOX[region],
     &     CHF_CONST_INT[startcomp],
     &     CHF_CONST_INT[endcomp])

      integer CHF_DDECL[i;j;k]
      integer nv

      dotprodout = zero

      do nv=startcomp,endcomp,1

         CHF_MULTIDO[region; i; j; k]

         dotprodout = dotprodout +
     &        afab(CHF_IX[i;j;k],nv)*
     &        bfab(CHF_IX[i;j;k],nv)

         CHF_ENDDO
      enddo

      return
      end
```

An equivalent Titanium method may look like

```
// T is an immutable class encapsulating
// state variables
public double DOTPRODUC(T [DIM d] local arrayA,
                       T [DIM d] local arrayB,
                       RectDomain<DIM> domain,
                       int startcomp,
                       int endcomp){

    double dotprodout = 0;

    for (int nv = startcomp; nv <= endcomp; nv++)
        foreach (point in domain)
            dotprodout+=
              arrayA[point][nv]*arrayB[point][nv];

    return dotprodout;
}.
```

Note that the spatial dimension in Chombo is a compilation-time constant, which means that you can build either a 2D or 3D Chombo code but not both. This limitation keeps Chombo from applications where computation may be performed on objects of different spatial dimensions. For example, in ocean modeling the horizontal dimensions (2D) are treated differently than the vertical one (1D). This is one of the motivations for our Titanium AMR project.

High-level abstractions supported at the language level gives Titanium a significant productivity advantage over the mixed-language approach. Meanwhile, it uses compiler and runtime optimizations to keep the performance constraint satisfied. Overall, the game for all parallel programming model efforts is to strike the right balance between ease-of-use and performance. We will discuss the performance and scalability issues of this benchmark in the next section. To compare the succinctness of these two programming approaches, Table 2 shows the lines of code (LOC) of the Titanium implementation and its Chombo counterpart.

We do not mean to use LOC as the solely measure of a language's expressiveness. In general, to evaluate the productivity of a programming system we need to take account of other factors besides performance, such as software quality (composability, maintainability, and reusability), turnaround time, and development cost. However, defining the right metrics for productivity and then developing the right

Component		Titanium AMR	Chombo
BoxTools	rectangular array	0	10781
	AMR data holder	1327	3996
	gridding & load balancing	840*	1547
	utility	311	4978
AMRTools		1081	2990
AMRElliptic		1717	4207

Table 2: Comparison of line counts of the two implementations (comments and empty lines are not counted). We try to decompose the functionality of the BoxTools library into four orthogonal parts: Chombo rectangular arrays, classes for defining data on unions of rectangular grids and mapping such data onto distributed memory systems, grid generator and load balancer, as well as utility classes. Chombo inherited the BoxLib rectangular array library developed at the Center for Computational Sciences and Engineering (CCSE) at LBNL. The Titanium implementation has different grid generator and load balancer. This fact is indicated by the * symbol. Overall, the Titanium implementation is significantly more compact than its Chombo counterpart.

procedure to measure these metrics is a challenging research question [13]. It is an important topic for our future work.

5. PERFORMANCE AND SCALABILITY

The computation and communication patterns of AMR calculations are more complex than those of ordinary finite difference schemes due to the irregularity introduced by local mesh refinement. In this section we introduce the performance challenges posed by this benchmark.

5.1 Serial Performance

Chombo uses Fortran to support high-performance array operations, whereas Titanium depends on its own compiler optimizations to achieve this goal. Some of those optimizations have been mentioned in Section 4. Our experience shows that the serial performance of the Titanium implementation matches that of Chombo. We have compared the two Poisson solvers on three platforms in solving the small test problem. The first one is a Pentium 4 workstation with a 2.8 GHz CPU (`seberg.lbl.gov`). The second one is a supercomputer named Seaborg at NERSC (National Energy Research Scientific Computing Center), which has 380 nodes for computation with 16 processors per node. Each Power 3 processor runs at a clock speed of 375 MHz. The nodes are interconnected by IBM Colony switches with two network adapters per node. The last one is another machine at NERSC named Jacquard, which has 356 dual-processor nodes for computation. Each processor runs at a clock speed of 2.2 GHz. The nodes are interconnected with a high-speed InfiniBand network. For more configuration details of these two machines, please refer to NERSC website (`www.nersc.gov`). The serial running times are summarized in Table 3. The grid configuration generated by Chombo is used by both solvers, which has been shown in Figure 2. For easier comparison we set the right-hand side f to zero

Platform	Titanium AMR	Chombo
Workstation	47.7	57.5
Seaborg	118.8	113.3
Jacquard	34.56	36.97

Table 3: Comparison of serial performance on three platforms. The running times shown here are in seconds.

and the initial value of ϕ to one for the performance study in this section. Thus, we know that the numerical solutions to the small and large test problems should converge to zero everywhere. Note that this simplification does not reduce the complexity of the numerical computations involved. All timing results in this section do not include the cost of grid generation, load balancing and initialization.

5.2 Load Balancing

A load-balancing algorithm in this context assigns a grid in the AMR hierarchy to a process. How a load balancer works directly affects the scalability of the entire computation. Note that there can be thousands of grids at one refinement level. Chombo's load balancer uses the Kernighan-Lin algorithm for solving knapsack problems [6, 15]. It tries to balance the computational load at each refinement level independently. The communication pattern as a result of this algorithm is illustrated in the left part of Figure 4, where the large test problem is solved with 42 processors (one process per processor) using the grid configuration generated by Chombo. The adjacency matrix shown here represents the communication relationship between processes. The dot at $[i, j]$ means process i needs information from process j. There are 1036 entries in this matrix and those not in the diagonal blocks indicate inter-node communications. You can see that this matrix is quite dense and far from being block-diagonal. The communication in block-structured AMR is indeed logically short-ranged – each grid only exchanges values with its neighbors. To utilize this locality property of AMR communication, we have implemented a load balancer in Titanium using Morton Space-Filling Curve also known as N-ordering, whose examples are shown in Figure 3. The locality advantage provided by N-ordering is obvious – the resulting adjacency matrix is sparser which has 705 entries and is better structured as shown in the right part of Figure 4. The less the entries means the more data transfers are performed locally to each process. Although the N-ordering load balancer still works on each refinement level independently, our experience shows that it improves the alignment between the top two finest levels (containing most of the data) so that communication between them is more likely to be intra-noded. The performance of other SFCs in this setting is subject to further investigation. Again, in general the goal is to make the adjacency matrix sparser and better-structured.

To leverage the better locality provided by the SFC load balancer, one optimization to do is to overlap local (intra-processor for MPI and intra-node for Titanium) data transfers with the global ones which are much more expensive. Better locality is also preferred when regridding so that less data is copied. The cost of this load-balancing algorithm is not expensive, which is dominated by sorting. We use Quicksort for our Titanium implementation.

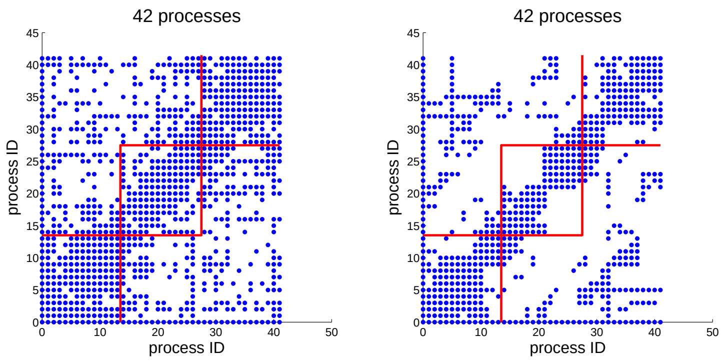


Figure 4: Communication patterns resulting from two load-balancing algorithms. The left graph shows the corresponding adjacency matrix from Chombo's load balancer which has 1036 entries, whereas the right one shows the adjacency matrix from the N-ordering load balancer which has 705 entries and is better structured. Here, the large test problem is solved with 42 processors using the grid configuration generated by Chombo. Suppose 3 nodes are used for the computation with 14 processors per node. Then the entries in the three diagonal blocks indicate intra-node communications, while the others indicate inter-node communications.

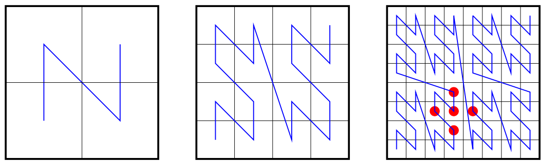


Figure 3: Examples of Morton SFC or N-ordering used in sorting the grids at each refinement level. Here, each square represents a grid cell. The order of each cell is determined by its location on the curve. In our load balancer, the order of each grid is set to be the order of its lower-left-corner cell at the next finer level in order to enhance level alignment.

5.3 Optimizing Communications

In AMR, a grid typically has more neighbors than in ordinary finite difference schemes and the number of neighbors varies from grid to grid. To optimize communication performance, Chombo packs and unpacks inter-process data transfers (into and from buffers) so that there is only one MPI message from one process to another during each communication operation, given that the global meta-data is made available to every process in Chombo. We can emulate this strategy in Titanium, but it will lead to an MPI programming style which is not natural for one-sided communication model. Unlike MPI where point-to-point synchronization is inherent in its semantics, Titanium does not support this kind of synchronization at the language level. To explore the right way to implement AMR in Titanium, we express data transfers as array copies at the logical level in this reference implementation and leave optimizations to the Titanium compiler and runtime. It is interesting to know how much the compiler can help to keep the performance constraint satisfied when communication is programmed at high level of abstraction.

While expressing communications as one-sided array copies is conceptually simple and much easier to program than using MPI, it results in fine-grained messages when implementing AMR. Although the scalability of the Titanium implementation is quite good in single SMP node as shown in Figure 5, it would not be sustained beyond one node if there were no optimizations done at all. When the number of nodes increases, so does the more expensive inter-node messages as indicated in Figure 6. The average size of array copies in the large test problem is around 4K bytes, implying that many of the data transfers are too small to achieve peak bandwidth on modern networks. Optimizations at the compiler and runtime level have been developed to perform automatic packing to aggregate small messages using an SMP-aware communication model [16]. The preliminary results indicate that this class of optimizations improve significantly the scalability of the Titanium implementation, even though the meta-data is not utilized as in Chombo. The Titanium Poisson solver can now scale similarly as Chombo does in solving the large test problem using up to 112 processors on Seaborg and Jacquard, although not outperforming it. (Preliminary experience with the Titanium implementation have been reported in [17].) The running times are summarized in Figure 7, where the grid configuration from Chombo is used and load-balanced by the N-ordering algorithm for both sides. The Titanium Poisson solver is on

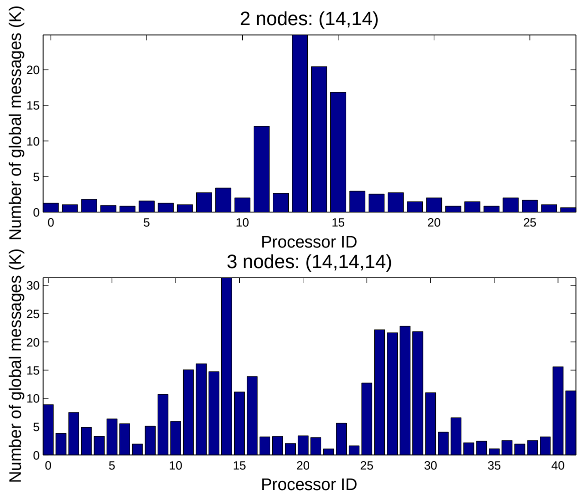


Figure 6: The distribution of global messages generated by the AMR exchange method in the Titanium implementation for solving the large test problem when two and three nodes are used, with 14 processors per node. More global messages occur at the node boundaries due to the locality property of communications in block-structured AMR.

average 10.6% slower than that of Chombo on Seaborg and 16.5% on Jacquard. Note that there is no SMP-aware optimization done in Chombo.

5.4 Utilizing Asynchronous Operations and Fine-grained Parallelism

In both Chombo and the Titanium implementation communications are bulk synchronous, where the communication phases are separated from the computation ones by global barriers. This style of parallel programming is common in scientific computing applications, particularly for implementing finite difference schemes. It is relatively easy to program bulk synchronous communications because global synchronization is conceptually simple. This practice is also consistent with the traditional wisdom of using MPI – all the data is available during the communication phase for packing so that a small number of large messages can be achieved. When this benchmark is implemented carefully (for example, by choosing the right load balancer) using the above programming model, we might be able to scale it up to ten thousands of processors (in terms of weak scalability). However, a great challenge facing everyone today is how to program an application and make it scale on the emerging peta-scale machines which may have an order of one hundred thousands processing units.

To make AMR scale further, finer-grained parallelism has to be exploited. In AMR, a grid may have different types of boundary conditions which can be handled in parallel, and applying a stencil operation in the inner region of a grid is independent of these boundary conditions. Particularly, exchanging ghost values which incurs most of the communication cost can be overlapped with computations. As shown in [2], one-sided communication models tend to have better performance than MPI in communicating messages of small to medium sizes, and overlapping communications with computations effectively spreads out the injections of messages into network which helps alleviate bandwidth competition especially on clusters where full bisection bandwidth is not available. Thus, if AMR is implemented in this way where more operations are performed in parallel and communications are spread out more evenly, then there is probably no need for packing messages at the application level as currently done in Chombo and hopefully we may be able to depend on optimizations at the compiler and runtime level to deliver good communication performance.

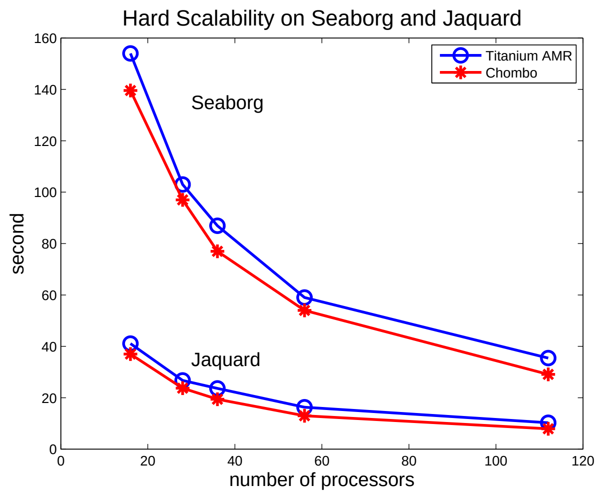


Figure 7: The hard scalability of the two solvers in solving the large test problem on Seaborg and Jacquard. The grid configuration is generated by Chombo's grid generator and load-balanced by the N-ordering algorithm.

The performance benefits of utilizing finer-grained parallelism in AMR such as overlapping communications with computations and relaxing global synchronizations are obvious, but the trade-off will be increased programming complexity. Any new programming language, designed for programming machines of large-scale hardware parallelism, should be able to express fine-grained logical parallelism in a conceptually easy way and provide adequate performance support at the compiler and runtime level. In the following pseudocode the concurrency in applying a stencil operation on an AMR grid is expressed using X10's activities. Note that X10's *region* is equivalent to Titanium's domain.

```
/*
 * async spawns an asynchronous activity
 * finish converts local termination of an activity
 * to global termination
 * Region and InnerRegion are of type region
 */
finish {
    // applying the stencil operation in the inner
    // region which is independent of boundary
    // conditions
    async for (point p: InnerRegion) stencilOp(p);
    // updating boundary conditions in parallel
    // where a grid may have three kinds of
    // boundary conditions
```

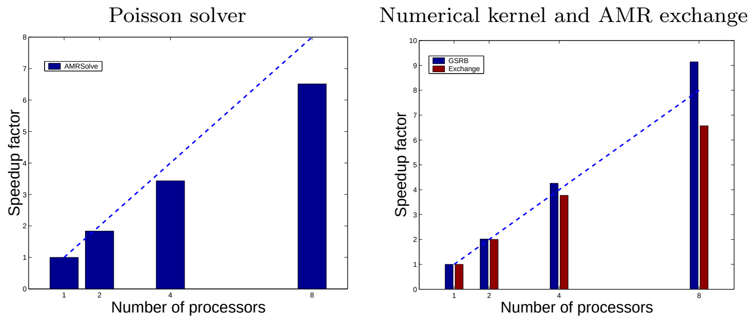


Figure 5: The hard scalability of the Titanium Poisson solver in solving the small test problem using the grid configuration from Chombo. The computation is performed in one node of Seaborg. The right figure shows the scalability of the numerical kernel and the AMR exchange method.

```
    finish {
        // for fine-fine boundary
        async exchangeGhostValues();
        // for coarse-fine boundary
        async quadraticInterp();
        // for physical boundary
        async physicalBoundary();
    }
    // applying the stencil operation on the rest
    // of region which depends on boundary
    // conditions
    for (point p: Region - InnerRegion)
                                      stencilOp(p);
}
```

6. CONCLUSION AND FUTURE WORK

We have devised an application benchmark for evaluating a parallel programming system's programmability and performance, and described the challenges posed by this benchmark through two reference implementations and in the context of three programming models. Unlike others, this benchmark imposes constraints on the quality of its implementation, such as composability, reusability and maintainability. Programming this benchmark with high performance and scalability is a very challenging task. The mixed-language approach, originally designed to ameliorate the programming difficulties posed by AMR algorithms particularly for traditional Fortran programmers, uses C++ for implementing high-level abstractions and irregular operations on arrays, while passing arrays to Fortran for high-performance bulk rectangular operations and programming message passing in a low-level style so as to achieve good communication performance. By supporting high-level abstractions at the language level, the Titanium implementation is more succinct and has more information explicitly available for compiler optimizations. We have experimented with fully depending on the Titanium compiler and runtime to optimize the performance of communication which is expressed at high-level abstraction of logic, and the preliminary result is encouraging. The goal here is to explore the right way to implement AMR productively on machines particularly of large-scale hardware parallelism. To achieve better scalability, parallelism at finer granularity has to be exploited. An X10 example is provided to show how fine-grained parallelism in AMR can be expressed at the language level. Ideally, a new parallel programming system can express applications such as AMR at high-level of abstraction and meanwhile provide sufficient support for performance optimization and tuning at the compiler and runtime level.

To make the definition of this benchmark complete, we still owe readers a set of metrics for measuring the productivity of a programming language. However, developing such a set of metrics is an open research question and it will be one topic for our future work. The latest news from Chombo is that by using the SFC load balancer and overlapping the intra-processor data transfers (memory copying) with the inter-processor MPI messages in the exchange of ghost values, Chombo is able to scale the AMR Poisson solver up to $8,000$ processors in terms of weak scalability (using different data set though). We will apply the same kind of optimization to the Titanium implementation where a larger degree of overlapping can be achieved because in Titanium we can do it at the node level rather than at the processor level. Particularly, we will continue to explore compiler and runtime optimizations for improving performance of the applications written in high-level of abstraction. An ambitious goal for us is to scale AMR Poisson up to $100,000$ processing units. People at IBM are interested in implementing this benchmark in the X10 programming language and having its implementation scale on machines such as Blue Gene/L. Hopefully, by studying this benchmark we can shed light on the search for new programming models suitable for programming machines of large-scale hardware parallelism such as the emerging peta-scale HPCSs.

7. ACKNOWLEDGMENTS

This project was supported in part by the Department of Energy under contract numbers DE-AC02-05-CH11231 and DE-FC03-01ER25509, the Department of Defense under contract number LB05001957, and the Defense Advanced Research Projects Agency (DARPA) under contract number NBCH30390004. Our thanks go to the Titanium programming language group at University of California, Berkeley for their support, and among them we especially want to thank Dan Bonachea and Amir Kamil for their valuable suggestions on the Titanium implementation of this benchmark. We also want to thank Daniel Martin at Lawrence Berkeley National Laboratory for his discussion of AMR Poisson solver and for providing us the initial test data sets. Lastly, we want to thank Vijay Saraswat at IBM T. J. Watson Research Center for his insights into the X10 programming language.

8. REFERENCES

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, D. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[2] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

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

[4] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. V. Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2005.

[5] Chombo website. `http://seesar.lbl.gov/ANAG/software.html`.

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

[7] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an npb experimental study. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2005.

[8] P. Hilfinger (ed.) et. al. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, University of California, Berkeley, 2005.

[9] P. Colella et. al. Chombo software package for AMR applications design document. Technical report, Applied Numerical Algorithms Group (ANAG), Lawrence Berkeley National Laboratory, 2003.

[10] FFTW website. `http://www.fftw.org`.

[11] High productivity computer systems website. `http://www.highproductivity.org/`.

[12] HDF5 website. `http://hdf.ncsa.uiuc.edu/HDF5/`.

[13] Research agenda for high productivity language systems. `http://www-unix.mcs.anl.gov/g̃ropp/bib/reports/hpl-research-agenda2.pdf`, 2004. Workshop Report, Argonne National Laboratory.

[14] D. Martin and K. Cartwright. Solving Poisson's equation using adaptive mesh refinement. Technical Report UCB/ERI M96/66, University of California, Berkeley, 1996.

[15] 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.

[16] J. Z. Su, T. Wen, and K. Yelick. Compiler and runtime support for scaling adaptive mesh refinement computations in Titanium. Technical Report UCB-EECS-2006-87, University of California, Berkeley, 2006.

[17] T. Wen and P. Colella. Adaptive mesh refinement in Titanium. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[18] X10 website. `http://x10.sf.net`.

[19] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *Parallel Symbolic Computation'07*, 2007.

[20] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen. Parallel languages and compilers: Perspective from the Titanium experience. *The International Journal of High Performance Computing Applications*, 21(2), 2007.

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