Compiler and Runtime Support for Scaling Adaptive Mesh Refinement Computations in Titanium

Jimmy Zhigang Su
Tong Wen
Katherine A. Yelick

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-87
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-87.html

June 13, 2006

Compiler and Runtime Support for Scaling Adaptive Mesh Refinement Computations in Titanium*

Jimmy Su, Tong Wen, and Katherine Yelick
UC Berkeley and LBNL

April 24, 2006

Abstract

In this paper we present a case study of implementing an Adaptive Mesh Refinement algorithm in Titanium, a partitioned global address space language based on Java. We extend prior work by focusing on the problem of scalable parallelism and demonstrate the use of language, compiler, runtime and application level support to address problems of load balancing, parallelism, and communication optimizations. We demonstrate that, while the Titanium code is an order of magnitude more compact than an equivalent program written in C++ and Fortran with MPI, the Titanium implementation has comparable seriable performance and scalability. The Titanium code uses one-sided array copy operations which are transformed by the compiler to perform automatic packing using an SMP-aware communication model suitable for hybrid shared and distributed memory machines.

1 Introduction

There is a constant tension in parallel programming model efforts between ease-of-use and high performance, with most programmers opting for conventional languages extended with a message passing library. In this paper, we present a comprehensive case study of programmability and performance of the Titanium language, which is a dialect of Java extended for parallel scientific computing (12). We choose a particularly challenging application, which is a Poisson solver written using Adaptive Mesh Refinement (AMR) and demonstrate that the Titanium language is more productive than C++/Fortran/MPI for this problem and achieves comparable performance. We have implemented a framework for block-structured AMR, which follows the design of the Chombo library (9) and covers a subset of its functionality. Chombo is a widely used AMR framework written in C++ and Fortran 77 with MPI for the support of parallelism. AMR algorithms expose the tension between programmability and performance in developing high-end scientific computing software. It contains irregular (hierarchical, pointer-based) data structures, input-dependent computational load which easily requires domain-specific load balancing, fine-grained communication for updating the boundaries of blocks in the adaptive mesh. We demonstrate that Titanium combines modern parallel programming language features, such as object-orientation and sophisticated array abstractions to enable productivity, while giving programmers sufficient control over key performance features, such as parallelism, data layout, and load balancing, to achieve high performance.

Our study builds on previous work by Wen and Colella (21)(22), where we demonstrated significant productivity advantages of Titanium over C++/Fortran 77 with MPI on AMR, and showed comparable serial

*This project was supported in part by the Department of Energy under contracts DE-FC03-01ER25509, KWC044, and 619501.

performance on workstations and parallel performance on a shared memory multiprocessor (SMP). In this paper, we explore the scalability of our implementation on a larger platform, such as a cluster of SMPs, and introduce optimizations at the application, compiler, and runtime levels to improve performance. We show that these approaches effectively increase the scalability and performance portability of our code, which is the same for both shared-memory and distributed-memory platforms. These techniques result in a Titanium AMR code that scales as well as Chombo does, and performs more sophisticated communication optimizations, but does so entirely with automatic support rather than hand optimizations. We also extend the previous Titanium AMR implementation with a load balancing algorithm based on space-filling curves, demonstrating the ability to use domain-specific knowledge in this critical aspect of performance.

The rest of this paper is organized as follows. Section 2 provides an overview of block-structured AMR, the Chombo package, and the Titanium language as way of background. Section 3 briefly introduces our Titanium implementation of AMR and summarizes the LOC (Line of Code) comparison between the two implementations. Section 4 describes the test problems and the grid configurations of our parallel performance study. Section 5 describes the main optimizations used by the Titanium implementation to enable scalable AMR performance; these are motivated by the fine-grained communication that naturally arises in AMR, but are not specific to the AMR code. Section 6 contains a performance study of the Titanium implementation, and quantifies the value of the optimizations performed by the Titanium compiler and runtime system.

As a matter of notation, we use the terms *grid* or *box* to refer to a rectangular mesh patch in the AMR grid hierarchy. We also use *thread* and *processor* synonymously; although multiple Titanium threads can be mapped to one physical processor, in this paper we assume the most common usage of Titanium in which this mapping is one-to-one.

2 Adaptive Mesh Refinement, Chombo and Titanium

2.1 Block-structured AMR

Since first developed by Berger and Oliger (2) and further refined by Berger and Colella (3), the Adaptive Mesh Refinement (AMR) methodology has been applied to numerical modeling of various physical problems which exhibit multiscale behavior. For example, solutions to Poisson's equation may have this type of variation due to local concentration of charges or boundary conditions. Although the idea of local refinement sounds straightforward, the algorithms involved are complicated. The complexity comes from the boundaries introduced by local refinement. In our approach to AMR, numerical solutions are defined on a hierarchy of nested rectangular grids. The grids with the same resolution, that is, with the same spacing (cell size), forms a level, and each level of grids are embedded in the next coarser one recursively according to certain rules (15) (16). In Figure 1, an illustration of two adjacent levels of grids is shown for the two dimensional case. In this example, three fine grids are superimposed on two coarse ones. Here, each solid rectangular box represents a grid patch, and each dotted square represents a grid cell. Thus, an array defined on the third fine grid, for example, will have 16 elements. Grids are sometimes accreted to accommodate ghost cells. Exchanging ghost values is a major source of communication. In a grid hierarchy, a fine grid is allowed to cover multiple grids at the next coarser level, and at each level, grids of different sizes may be used. Passing information from a fine grid to a coarse one below it and vice versa is another source of communication. This patch approach is different from the tile one in that grid tiles do not overlap with each other as grid patches do.

In general, there are three kinds of operations involved in our AMR implementation:

1. Local stencil operations on grids such as those in an iterative solver

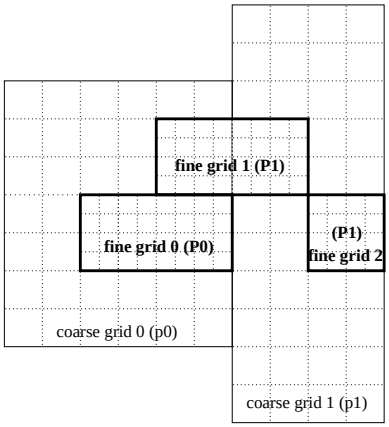


Figure 1: Two adjacent levels of grids are shown in this example for the two dimensional case. Each solid rectangular box shown here represents a grid, and the dotted squares inside a box are the cells it contains. There are two grids at the coarse level and three at the fine level. For each grid, the number in parenthesis represents the processor it is assigned to. The refinement ratio between these two refinement levels is 2 in every space dimension.

2. copying from one grid to another
3. Irregular computations on grid boundaries, for example, the one to interpolate the ghost values at the coarse-fine grid interfaces

Due to the irregular computation and data access mentioned above and the complicated structures controlling the interactions between levels of refinement, it is challenging to implement AMR algorithms efficiently, especially in parallel. In this paper, the details of the AMR algorithms we have implemented are not covered, but they are well documented in (15) (9) (21). Here, all algorithms are cell-centered, and the refinement ratio is uniform in each space dimension.

2.2 Chombo

We have implemented a subset of Chombo's functionality in the Titanium language. Chombo is a widely used AMR framework written in C++ and Fortran 77, with MPI for the support of parallelism. The execution model of Chombo is SPMD with bulk synchronization, where the stages of communication and computation are alternated. Chombo uses Fortran 77 for the support of multidimensional arrays, given that Fortran is a better compromise between expressiveness and performance than C/C++ for multidimensional arrays (but still not entirely satisfactory: no dimension independent syntax). Numerically intensive operations on grids are implemented in Fortran 77 for better performance, whereas memory management, control flow, and irregular operations on grid boundaries are implemented in C++. In Chombo, C++ templates are used extensively, and class inheritance is used mainly to define interfaces. From Chombo's experience, the downside of the mixed-language approach is high maintenance and debugging cost.

2.3 Titanium

The Titanium language is an explicitly parallel dialect of Java for high-performance scientific programming (23) (24). Compared to conventional approaches to parallel programming such as MPI, Titanium allows more concise and user-friendly syntax, and makes more information explicitly available to the compiler. The goal of Titanium is to achieve greater expressive power without sacrificing parallel performance.

Titanium's model of parallelism is SPMD with Partitioned Global Address Space (PGAS). In the following, listed are the key Titanium features added to Java, which are covered in our implementation of AMR:

1. **Titanium arrays** - Titanium's multidimensional arrays with global index space provide high-performance support for grid-based computations, 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, a cell as shown in Figure 1 is indexed by an integer point, and a domain is a set of these points which can be either rectangular or not. Points (`Point`) and domains (`RectDomain, Domain`) are first-class types and literals.

3. **Foreach loops** - In Titanium, the iteration over any multidimensional Titanium array is expressed concisely in one loop, using its unordered looping construct `foreach`. For example, a loop over a 3D array may look like the following:

   ```
   final Point<3> low = [-1,-1,-1], high = [2,2,2];
   final RectDomain<3> RD = [low:high];
   double [3D] TA = new double [RD];
   //RD can be replaced with TA.domain()
   foreach (p in RD) TA[p] = 1.0;
   ```

   In contrast, Fortran has to use a nested `for` loops to iterate over multidimensional arrays, and the dimension (rank) of the index can not be hided.

4. **Global data structures** - In Titanium, the parallelism comes from the distribution of data. It is each thread's responsibility to construct the local piece of a global data structure and have its reference available to others. Global data structures are built in Titanium through its general pointer-based distribution mechanism.

5. **One-sided communication** - Each thread can access a global data structure as if it was local. The bulk communication between two threads is realized through the `copy` method of Titanium arrays. For example, let `TAdst` and `TAsrc` be two Titanium arrays that may reside on different processors, copying `TAsrc` to `TAdst` is expressed concisely as

   ```
   TAdst.copy(TAsrc);
   ```

6. **Templates** - Titanium's support of templates like those in C++ proves useful in our AMR implementation, where generic parameters may be either constant expressions or types (including primitive types).

7. **The local keyword and locality qualification** - Titanium's `local` type qualifier enables the expression and inference of locality and sharing properties of a global data structure. Locality information is automatically propagated by 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 (value) classes. Operator overloading allows one to define custom operators applied to user-defined classes.

In Section 5, we present a compiler optimization for aggregation of array copies. The code transformation done by the optimization must obey the Titanium memory model. Here are some informal properties of it.

Component	Titanium	C++/Fortran 77/MPI
Base layer	2400	35000
Middle layer	1200	6500
Elliptic PDE solver	1500	4200

Table 1: As a rough comparison of language expressiveness for AMR applications, this table compares the line counts of our Titanium benchmark with those of the corresponding Chombo implementation. The increase in the line count for the base-layer implementation, compared with the number of 2000 reported in (21) is due to the adding of a fixed-size grid generator and a load balancer using Space Filling Curves. The overall Chombo package has $O(10^5)$ lines of code.

1. **Locally sequentially consistent** - All reads and writes issued by a given processor must appear to that processor to occur in exactly the order specified. Thus, dependencies within a processor stream must be observed.

2. **Globally consistent at synchronization events** - At a global synchronization event, such as a barrier, all processors must agree on the values of all the variables. At a non-global synchronization event, such as entry into a critical section, the processor must see all previous updates made using that synchronization event.

For more details of Titanium, readers please refer to (12).

3 Titanium's AMR implementation and its productivity

Our Titanium implementation of block-structured AMR follows Chombo's design of software architecture. The motivation of this project is to provide a comprehensive case study of Titanium's expressiveness and performance, meanwhile to build a framework for expediting the development of AMR applications by leveraging the productivity strength of Titanium. Chombo uses layered design to maximize code reusing and to hide details of parallelism. There are three layers of functionality in Chombo's software architecture:

1. Abstractions such as those provided at the language level in Titanium: points, domains, and multidimensional arrays with global index space, as well as parallel data structures defined on a union of disjoint rectangular domains

2. Operators implementing the coupling between refinement levels such as the one that interpolates the ghost values at the coarse-fine grid interfaces

3. Modules for elliptic PDEs, time-dependent adaptive calculations and particle-in-cell methods

The first two layers provide fundamental support for AMR applications, on which PDE solvers are built with the modules from the third layer. Overall, the three layers provide a framework for developing AMR applications.

We have implemented most of the functionalities defined at the first two layers, as well as the elliptic solver module at the top one. In our implementation, the classes at each layer are organized into a Java package. As a simple measure of productivity, the numbers of LOC from two implementations are compared with each other. The comparison is summarized in Table 1. For more details of our Titanium implementation, readers please refer to (21) (22).

Titanium code tends to be more concise and expressive. Compared with the corresponding codes in Chombo, our Titanium codes are shorter, cleaner and more maintainable. Titanium's syntactic support for grid-based computations and its PGAS programming model with one-sided communication ease our programming effort significantly in implementing AMR. Particularly, Titanium enables dimension independent programming. In Chombo, all objects have to be in the same dimension due to Fortran's lack of dimension independent syntax, which has limited Chombo's application domain. Another important feature of the Titanium compiler is that it automatically aggregates messages. From the performance point of view, message aggregation is the key approach to achieve good scalability on distributed-memory platforms. The resulting productivity gain is that programmers need not pack and unpack messages as they typically do in writing a scalable MPI code. In our implementation, communication is done only through array copies without further packing. Titanium's prevention of deadlocks on barriers is also a helpful productivity feature.

4 Test problems

We test our implementation by solving Poisson's equation

$$L\phi = f \text{ in } \Omega, \tag{4.1}$$

subject to Dirichlet boundary condition

$$\phi = g \text{ on } \partial\Omega. \tag{4.2}$$

Here, L is the Laplacian operator, that is, $L = \Delta$, and Ω is an open cubic domain of unit size. The lower left corner of Ω is $(0, 0, 0)$.

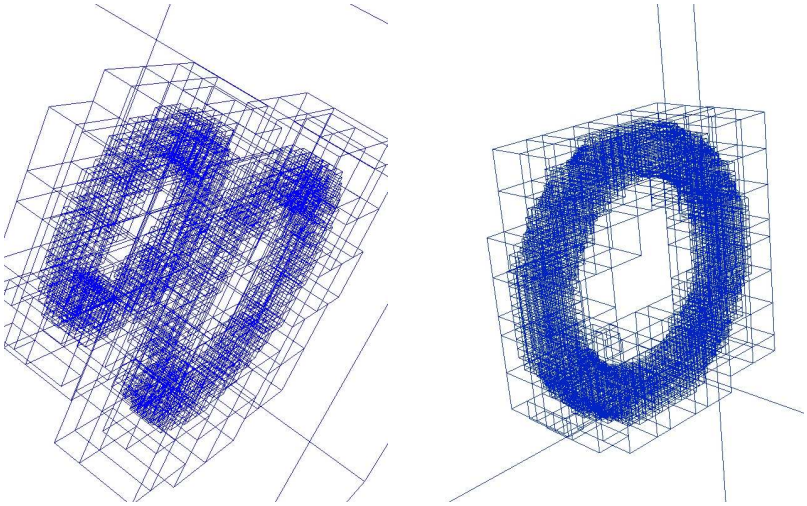
Two setups of the above problem are used in the following performance studies - one for small runs and the other for large ones. The small test problem has two vortex rings in its problem domain, while the large one has only one ring. The two corresponding grid configurations are imported from Chombo, which are shown in Figure 2. The grids in the test problems are static, that is, they do not change during the computation.

For the purpose of easier comparison and without loss of generality, we set functions f in 4.1 and g in 4.2 to zero and the initial value of ϕ to one. 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 computation involved. The convergence criteria is

$$||\rho_m||_\infty \leq 10^{-10}||\rho_0||_\infty, \tag{4.3}$$

where ρ_i is the composite residual at step i.

Each level of grids are load-balanced independently in both Chombo and our Titanium implementation. In the following performance study, we use our own Space Filling Curve (SFC) load-balancer (22) instead of Chombo's heuristic Knapsack one (16) for both sides. Our load-balancing algorithm exploits SFC's locality-preserving property by using one to order the grids. Once the grids are ordered and sorted, a greedy algorithm is used to assign each grid to a processor so that the total grid volume (number of cells) is balanced. The resulting algorithm is very simple with complexity of $O(n \log n)$ in average, where n is the number of grids. Figure 3 shows three examples of Morton or N-ordering SFC that we use for our load balancer. Here, the Morton order of the five marked cells is South, West, Center, North, and East. Our observation is that the SFC load balancer tends to result in less inter-node array copies on a distributed memory platform, such as a cluster of SMPs (22). However, when message packing/unpacking and aggregation are performed, we expect the performance difference between these two algorithms is small.



Small configuration			Large configuration		
Level	**grids**	**cells**	**Level**	**grids**	**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. In the columns labeled by "grids" and "cells", listed are the numbers of grids and cells at each level respectively. 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 dimension for both cases.

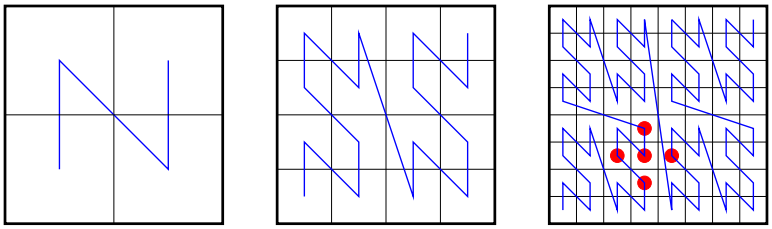


Figure 3: Examples of Morton or N-ordering SFC used in our load balancer to order the grids at each level. Here, each square represents a grid cell. The order of each cell is determined by its location on the curve. In our load balancer, the order of each grid is set to be the order of its lower-left-corner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

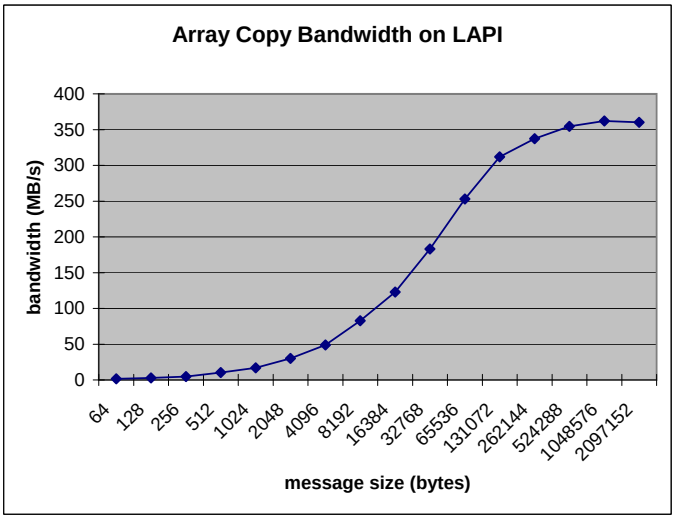


Figure 4: Bandwidth on LAPI for different message sizes

5 Optimization

Communication is implicit in Titanium through reads and writes to fields and array elements, and array copies. This can lead to significant reduction in code size in comparison to programming with MPI, because Titanium programmers do not need to explicitly handle packing and unpacking to and from buffers. While this dramatically increases the programmability of Titanium, it leads to many small messages if the Titanium code is compiled in a straight forward way. On most modern networks, large messages achieve much greater bandwidth than small messages. Figure 4 illustrates the case on the LAPI network, which we use in our experiments.

All the communications within and between refinement levels in our Titanium implementation are realized through array copies. These array copies often require packing and unpacking, because their intersection may not be contiguous in memory. If the intersection domain in the source array is non-contiguous, then packing is required. Analogously, if the intersection domain in the destination array is non-contiguous, then unpacking is required.

The intersection size for these array copies in AMR is between 512 bytes and 8K bytes. Looking at Figure 4, we do not get peak bandwidth by sending individual messages of those sizes. Compiler support is added to the Titanium compiler to generate code automatically to do packing and unpacking, and aggregate messages over multiple array copies. This allows the programmer to write the Titanium code in a straight forward way, and still get the performance boost from message aggregation.

For example, in the large test problem, the array copy method is called 281556 times in exchanging ghost values. Each call to array copy for the large test problem on average retrieves 4.24K bytes of data, which is too small to achieve peak bandwidth on the network.

Ideally, we would like to queue up the array copy requests and send many of them at a time. This would reduce the number of messages, and allow the data retrieval to obtain higher bandwidth. In our optimization, we are able to overlap both the packing request and the data retrieval with other communications. Further

aggregation is obtained by combining requests for the same SMP node. Both will be described in the following section.

5.1 Compiler Transformation

Based on the control flow graph, a Titanium program can be divided into phases by barriers. The optimization looks for aggregation opportunities within a phase. Array copies can not be aggregated across barriers, because an array copy consists of both reads and writes, and a read before a barrier can not see changes due to writes after the barrier. The transformation looks for array copies where the destination array is known to be local at compile time, because the optimization is pull based. It can be extended to push based, and this will be one of our future works. Programmers can declare an array to be local or the compiler can automatically find local pointers using Local Qualification Inference (14). Using def/use analysis, we find the least common ancestor of all the uses of the destination arrays in the dominator tree, and insert a flush call there to the runtime system to ensure all the array copy calls are completed by that time. The uses referred to here do not include the array copies to be aggregated. It is important to note that the runtime system is free to retrieve some of the intersection data before flush is called. The flush call only ensures completion.

The optimization is inter-procedural, where the array copies from different methods can be aggregated and the flush call can be pushed down past method boundaries. In the execution path between the array copy to be aggregated and the flush call, we restrict access to the source and destination arrays to other aggregate array copies. This is checked statically at compile time. This condition ensures that the transformation obeys the first rule of the Titanium memory model, where a processor must be able to see its own writes. This restriction can be relaxed if we add runtime checks to detect when the source or destination arrays are accessed, and call flush when this occurs. In practice, our Titanium applications do not favor the latter approach. The Titanium code is separated into computation and communication phases separated by barriers. The extra overhead of runtime checks is not justified.

We can see from Figure 4 that bandwidth reaches more than 80% of peak at message size of 100K bytes. The benefit of aggregation diminishes beyond that size due to the need for bigger buffers. A threshold is set at the runtime system for executing the queued up array copies when the aggregate message size reaches the threshold. We set it to 100K bytes for the LAPI network. We can set this number at compiler build time by executing the benchmark in Figure 4. It is our belief that machine specific optimizations such as this should be handled by the compiler, so that the application code does not need to change due to switch of hardware.

5.2 SMP Aware

On a cluster of SMPs, the processor configuration may use multiple processors on a node. In this case, we combine multiple array copy requests where the source arrays belong to the same node. The source arrays may be owned by different processors on the same node. The processors on the same node share the same address space if they run under the same process, so any processor on that node in the same process can access those source arrays. Packing requests are handled inside of an Active Message handler. The packed data is retrieved using non-blocking gets on the requesting node. This allows overlap of communication with communication.

The retrieved intersection data are unpacked in the same order as the sequence of array copy calls. This way it is OK for the same array element to be written more than once. The element will have the last write value as the unoptimized version.

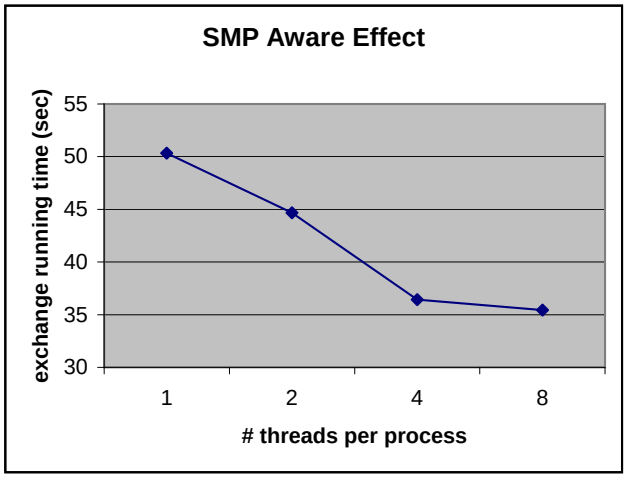


Figure 5: Effects of SMP aware optimization on the exchange method by varying the number of threads within a process. Experiment was done using 16 processors on 2 nodes on the IBM SP machine.

6 Performance Study

We use Seaborg from NERSC for our experiments. Seaborg is an IBM RS/6000 SP machine. Its network uses SP Colony Switch 2 running LAPI. The CPUs are 375 MHz Power 3+ processors. We will refer to Seaborg as the IBM SP machine from this point on in the paper.

Figure 5 illustrates the effect of the SMP aware optimization for the exchange method in AMR, where the values in the intersections between grids on the same level are copied. We use two nodes and 8 processors per node for this experiment. We vary the number of processes and the number of threads per process in the experiment. Increasing the number of threads per process increases the opportunity for aggregation, because threads in the same process can access the same address space. We adjust the number of total processes accordingly to keep the number of physical processors at 16.

The running time of exchange improves by 30% as we increase the number of threads per process from 1 to 8. The improvement is due to the reduction in the number of remote messages. The 8 threads per process case sends 9744 remote messages. The 1 thread per process case sends 13939 remote messages. That is a saving of 30%. It is important to note that for two processes on the same node, although they don't share the same address space, messages between them do not incur a network round trip, so those messages are not included in our statistics.

We run both the Titanium AMR and Chombo on the same processor configurations for performance comparison. We use the small data set for measuring serial performance. On the IBM SP machine, it took 118.8 seconds and 113.3 seconds to solve this problem for Titanium AMR and Chombo respectively. The difference is 4.9%. The serial performance difference between Titanium AMR and Chombo depends on the machine. For the same problem size, on an Intel Pentium 4 (2.8 GHz) workstation, it takes Titanium AMR 47.7 seconds to solve the problem. In comparison, it takes Chombo 57.5 seconds on the same machine.

For parallel runs, we use the large data set with various processor configurations from 16 processors to 112 processors. The problem is too large to run on a single processor. Figure 6 shows the speedup graph comparing Titanium AMR with aggregation, Titanium AMR without aggregation, and Chombo. We assume

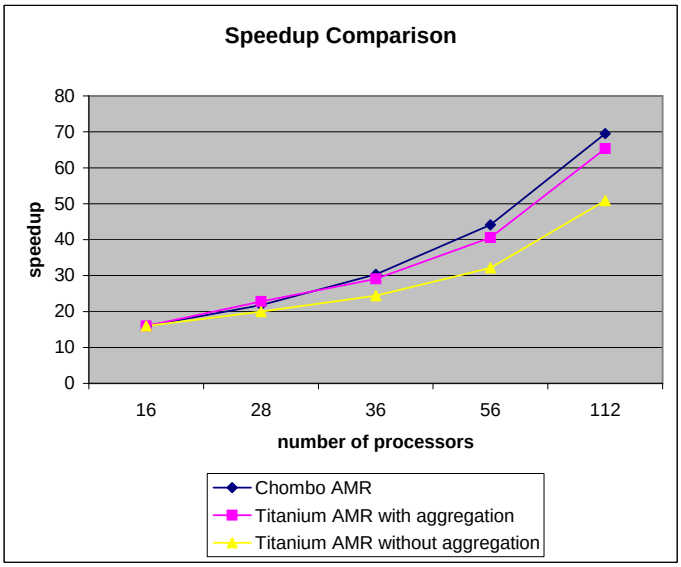


Figure 6: Speedup comparison between Chombo, Titanium AMR with aggregation, and Titanium AMR without aggregation. Experiments done on IBM LAPI machine with the large data set.

linear speedup up to 16 processors in the graph. Titanium AMR with aggregation can achieve similar speedup to Chombo up to 112 processors. Without aggregation, Titanium AMR does not scale nearly as well due to the increasing number of remote messages as more processors are added. More neighboring grids become remote as the number of processor increases. Titanium AMR with aggregation is on average within 10.7% of Chombo in terms of absolute performance.

7 Related Work

There are several AMR libraries that are widely used, such as Cactus (6), CCSE Application Suite (7), Chombo, and SAMRAI (17). Our approach is different from the library-based ones in that the compiler relieves from programmers the burden of arranging communication to achieve scalability. The support for multi-dimensional arrays and domains also allows the AMR benchmark to be written more concisely in Titanium.

An array copy A.copy(B) can be viewed logically as A[p1]=B[p1], A[p2]=B[p2], ... , A[pn]=B[pn], where p1, p2, ... , pn are the points in the intersection domain of A and B. With this view, we can look at the array copy aggregation optimization as a special form of the inspector executor optimization. Inspector executor is an optimization technique for indirect array accesses such as A[B[i]], where A is the remote array and B is the index array. In the inpector phase, the array address for each A[B[i]] is computed. After the inpector phase, the needed data is fetched over the network. And finally in the executor loop, values for each A[B[i]] are read out of the local buffer. Instead of fetching each element of A[B[i]] individually, the communication is done in bulk to reduce the number of messages.

In array copy aggregation, the code path before the flush call can be viewed as the inspector, where we find out the intersections for each array copy, analogous to finding out the indirect array accesses in the classical inspector executor optimization. After the flush call, the needed data have been unpacked to their destination arrays. They are subsequently used when the destination arrays are accessed. This can be viewed as the executor.

There has been substantial amount of work done in the inspector executor line of research. Walker proposed and implemented the idea of using a pre-computed communication schedule for indirect array accesses to distributed arrays in a Particle-In-Cell application (25). The idea is widely used in application codes today, where it is programmed manually. Use of the technique with compiler and library support was developed by Berryman and Saltz. The PARTI runtime library (5) and its successor CHAOS (18) provided primitives for application programmers to apply the inspector executor optimization on the source level. Our technique is automatic, and does not need annotation from the programmer. The ARF (26) and KALI (13) compilers were able to automatically generate inspector executor pairs for simply nested loops. More recently, our Titanium compiler added support for inspector executor style optimization based on communication models (19). Basumallik and Eigenmann applied the inspector executor optimization in the translation of OpenMP programs to MPI programs (1). Neither of those two approaches are SMP aware.

8 Conclusion

This paper uses a case study in AMR to show several advantages of using high level languages, Titanium in particular, for programming parallel machines. Our results show that, while the Titanium implementation is roughly an order of magnitude smaller than an equivalent portion of the Chombo AMR framework implemented in C++/Fortran with MPI, both serial and parallel performance is comparable. Titanium carefully balances the features of the machine that can be hidden from programmers, while giving them control over machine resources necessary for performance. The use of a domain-specific load balancing algorithm demonstrates the users' ability to control data layout, even for the very irregular AMR data structures, and load balancing.

Nevertheless, the compiler plays a central role in acheiveing scalability. The parallel performance results rely on communication optimizations, which perform communication aggregation that is SMP-aware and therefore appropriate for the popular hybrid machines that combine shared and distributed memory. The results highlight the advantages of using a language approach, rather than one based purely on libraries. The complexity of Titanium arrays are largely hidden in the runtime and compiler, allowing this investment to be re-used across application frameworks; the array abstractions provide mechanism for computing sub-arrays and copying, but are not specific to AMR and have proven useful in other numerical methods. Whereas heavily abstracted library code is often opaque to an optimizing compiler, the availability of a translation phase in a language-based programming model allows the compiler to perform sophsticated global optimizations, including the communication aggregation done here; this relieves application and framework programmers from tedious and error-prone code to pack and unpack data and enables machine-specific optimizations that would greatly add to maintenance costs if done manually.

References

[1] A. Basumallik and R. Eigenmann. Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems. *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2006.

[2] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53:484-512, 1984.

[3] M. Berger and P. Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of Computational Physics*, 82(1):64-84, 1989.

[4] 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.

[5] H. Berryman and J. Saltz. A Manual for PARTI Runtime Primitives, 1990

[6] Cactus Code Website: `http://www.cactuscode.org/`.

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

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

[9] P. Colella et. al. Chombo Software Package for AMR Applications Design Document. Applied Numerical Algorithms Group (ANAG), Lawrence Berkeley National Laboratory, Berkeley, CA, 2003.

[10] K. Datta, D. Bonachea, and K. Yelick. Titanium Performance and Potential: an NPB Experimental Study. *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2005.

[11] E. Givelberg and K. Yelick. Distributed Immersed Boundary Simulation in Titanium To appear *SIAM Journal of Computational Science*, 2006.

[12] P. Hilfinger (ed.) et. al. Titanium Language Reference Manual, Version 2.19 Technical Report UCB/EECS-2005-15, UC Berkeley, 2005.

[13] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting Shared Data Structures on Distributed Memory Machines. *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 1990.

[14] B. Liblit and A. Aiken. Type Systems for Distributed Data Structures. *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2000.

[15] D. Martin and K. Cartwright. Solving Poisson's Equation Using Adaptive Mesh Refinement. Technical Report UCB/ERI M96/66, UC Berkeley, 1996.

[16] 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.

[17] SAMRAI Website: `http://www.llnl.gov/casc/SAMRAI/`. Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory, Berkeley, CA.

[18] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. *Proceedings of Supercomputing*, 1994.

[19] J. Su and K. Yelick. Automatic Support for Irregular Computations in a High-Level Language. *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[20] Titanium Website: `http://titanium.cs.berkeley.edu`. Computer Science Department, University of California, Berkeley, CA.

[21] T. Wen and P. Colella. Adaptive Mesh Refinement in Titanium. *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[22] T. Wen, P. Colella and K. Yelick. An Adaptive Mesh Refinement Benchmark in Titanium. Submitted to *ACM TOMS*, 2006.

[23] 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.

[24] 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. Submitted to *International Journal of High Performance Computing Applications*, 2005.

[25] D. Walker. The Implementation of a Three-Dimensional PIC Code on a Hypercube Concurrent Processor. *Conference on Hypercubes, Concurrent Computers, and Application*, 1989.

[26] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed Memory Compiler Design and Sparse Problems, 1991.