Optimization and Evaluation of a Titanium Adaptive Mesh Refinement Code

Amir Kamil Ben Schwarz Jimmy Su
kamil@cs.berkeley.edu bschwarz@cs.berkeley.edu jimmysu@cs.berkeley.edu

May 19, 2004

Abstract

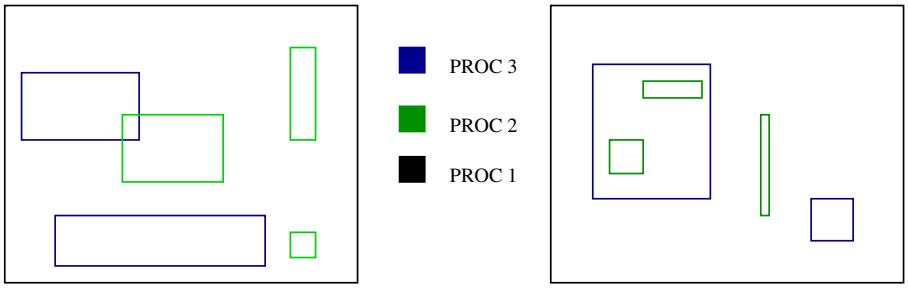
Adaptive Mesh Refinement (AMR) is a grid-based approach to approximating the solutions to partial differential equations (PDEs). It is governed by the principle that some portions of the grid require more computation to solve than other regions. AMR uses an adaptive approach to adjust the resolution of the computational domain to reflect the varying computational needs. A high-performance implementation of an AMR code is provided by the Chombo framework—a set of tools for implementing the finite difference method to solve PDEs. Chombo provides a C++ implementation that uses the Message Passing Interface (MPI) for communication.

Titanium is a Java-based language for parallel programming that uses a shared memory space abstraction. It offers many features to assist a developer in crafting parallel programs. However, the ease of use can often come at the cost of performance. In this paper, we evaluate an implementation of AMR provided by the Applied Numerical Algorithms Group at the Lawrence Berkeley National Laboratories. We explain several optimizations that we have performed on the code to improve its performance. Our contribution is an optimized implementation that runs 23% faster than the original Titanium code. It is either comparable to or faster than the C++/MPI Chombo code on all the platforms we have tested.

1 Introduction

This paper explores optimizations and analyses carried out on an AMR code that is implemented using the Titanium programming language [4]. Titanium provides a shared memory abstraction and many parallel programming constructs to simplify programming scientific codes. Unfortunately these features sometimes come at the price of inefficiency. For example, Titanium allows a program to have both local and remote pointers; remote pointers point to objects residing on another machine. Pointer operations, such as dereferencing, can be very expensive, because at runtime a number of checks need to be performed, and possibly communication needs to be done. In order to write an efficient program, the software developer using this abstraction must be aware of the intricate details of the underlying language framework. Our contribution is to analyze a Titanium implementation of AMR and optimize the constructs that are the sources of inefficiency.

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 explains the basic AMR algorithm along with some important data structures. Section 4 discusses the analysis we have performed on the initialization code—which accounts for a large percentage of the overall running time—and several optimizations performed to it. Section 5 lists our profiling results. A load balancing study is explained in Section 6. Our optimizations to the exchange method of the AMR code is described in Section 7.2. The final performance data is presented in Section 8, and conclusions are in Section 9.



Communication: Exchange of Borders

Communication: Interpolation

Figure 1: Sources of Communication in AMR

2 Related Work

The Titanium programming language provides the basis for many of our optimizations [4, 2]. Titanium is a dialect of Java, an object-oriented language, however the AMR is not written using an object oriented paradigm. AMR++, on the other hand, provides an object-oriented design for the problem [3]. Chombo is a high-performance AMR code provided by the Applied Numerical Algorithms Group at the Lawrence Berkeley National Laboratory [1].

3 Synopsis of AMR

AMR is a grid-based approach to approximating the solutions to PDEs. The grid is subdivided into smaller grids when a finer resolution is needed. A grid level refers to a set of regions with the same resolution; these regions are rectangular, and may overlap slightly. When refinement is done, we increase the resolution and divide the region into a set of smaller region. The smaller regions need not include all of the larger region. In particular, areas of the larger region which need not be further refined are not broken down. At any given level, the set of regions are distributed among the processes. It is not a requirement that a processor contain all the finer resolution regions within a larger box. The AMR is carried out by doing multigrid V-cycles on all the regions.

There are two sources of communication in the AMR code, as shown in Figure 1. The first is when two regions are near each other, and to perform the computation they need to retrieve values from each other. We shall refer to this type of communication as exchange of shared borders. In the figure, processor 2 and processor 3 share a small region, and the overlapping part represents the exchange communication that will take place. The second source of communication is when a finer resolution region is contained within a region with a larger resolution, and they are owned by different processors. A basic operation in the multigrid method is interpolation between resolutions. More specifically, the region layout is constructed top down starting with the most coarse grid. If a finer grid is needed, the coarse grid points are interpolated to figure out the values for the smaller region. Vice-versa, coming "up" on the multigrid v-cycle, interpolation needs to be done to set the values of coarser grid points based on the finer resolution points. This bidirection interpolation results in communication between the processors containing those grids.

4 Initialization Tuning

In the initialization stage a distributed data structure is built. The data structure consists of all the box layouts and their distribution among processors. Figure 2 shows a sample data structure for a simple two processor case. Note that each processor has a pointer to the data on every other processor. The intuitive way to implement this is to have each processor broadcast its pointers to the other processors. In practice, however, broadcasts must proceed serially so this can be a limiting factor. An alternative approach is to exchange pointers to the top level of

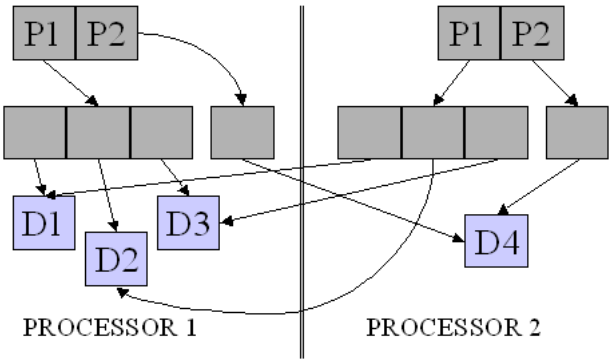


Figure 2: Example of Data Structures Built During Initialization

pointers and then use Titanium's array copy mechanism, which can proceed in parallel. We have evaluated both implementations and the results are presented in Table 1. The experiment was run using different number of processors on a single node. Surprisingly, the simpler broadcast is more efficient in practice. We speculate that using multiple nodes would favor the exchange method, since communication actually requires going over the network in that case.

5 Profiling Analysis

Much of the AMR Titanium code scales well as the number of processors increases, such as GSRB as shown in Figure 2. However, some pieces scale poorly or not at all: the initialization and CFInterp in particular. We profiled these two sections using timers and PAPI counters in order to determine the cause of this slowdown.

We first identified the pieces of the code in the initialization and CFInterp that ran slowly. In the initialization, three lines of code (two in `QuadCFInterp.ti:define()` and one in `LevelFluxRegister.ti:define()`) all consisting of `Domain-RectDomain` subtractions were the main culprits. In the first CFInterp (CFInterp1), the slow piece of code was a loop in which `double` operations occur. The picture was not as clear for the second CFInterp (CFInterp2); the slowdown appeared to be spread out across a large loop.

Purely by chance, we stumbled on a few possible explanations for the poor scaling in initialization and CFInterp. The AMR code is intended to be run with multiple Pthreads spawned from a single Unix process in order to allow shared memory communication. By accident, we initially ran the code with multiple Unix processes instead, with a single Pthread per process. As would be expected, the exchange operation slowed drastically as shown in Figure 2 since it is a communication step. To our surprise, however, the initialization and CFInterp operations ran much faster with multiple processes; in fact, the first CFInterp (CFInterp1) scaled perfectly with processes, though it did not scale at all with Pthreads. Two possible explanations came to mind: higher cost of static accesses with Pthreads and false sharing.

5.1 Static Accesses

Static accesses with Pthreads are expensive for two reasons. The first is that since Pthreads share the heap segment of memory, an expensive thread ID lookup must occur in order to determine which static variable belongs to a particular thread. Secondly, the copies of a static variable corresponding to each thread are stored contiguously, i.e. thread one's copy of the variable is stored next to thread zero's, and thread two's next to thread one's. This results in false sharing when multiple threads simultaneously access a static variable. The results of a microbenchmark comparing instance and static variables with processes and Pthreads are shown in Figure 3. Even with

Processor	*Execution Time (seconds)*		Change
	Exchange	Broadcast	
1	40.5	40.74	-0.24
2	52.06	51.93	-0.13
4	65.4	65.78	0.38
8	89.26	87.09	-2.17
16	138.5	105.0	-33.5
Geometric Mean:			-4.07

Table 1: Broadcast and Exchange Comparison

Time in Seconds (processes/Pthreads per process)

Operation	1/1	2/1	1/2	4/1	1/4	8/1	1/8
SolveAMR	235.4	164.0	140.3	147.3	91.05	128.3	87.47
Exchange	84.16	84.68	48.1	97.46	31.64	99.05	41.63
GSRB	59.97	29.69	30.58	14.69	15.29	7.22	7.12
CFInterp	45.4	24.28	35.02	16.03	29.61	10.1	30.61
CFInterp1	25.78	12.74	20.65	5.73	18.5	3.23	18.85
CFInterp2	19.62	11.54	14.37	10.3	11.11	6.87	11.76
Initialization	41.68	31.6	53.82	27.04	67.61	29.91	92.17

Table 2: Running time for AMR operations using processes and Pthreads.

Time in Seconds (processes/Pthreads per process)

	10M accesses		100M accesses	
	8/1	1/8	8/1	1/8
Instance	0.0342	0.0344	0.335	0.335
Static v2.405	1.68	11.6	16.8	115.4
Static v2.409	0.347	0.347	3.48	3.47
Indirect	0.241	0.241	2.41	2.42

Table 3: Static, instance, and indirect accesses using processes and Pthreads.

processes, a large slowdown over instance variables is experienced with static variables.

As Figure 3 indicates, one possible solution to the slowdown of static accesses that still provides sharing among objects is to use indirection. For example, if an integer needs to be shared among the objects of a class, a static final integer array of size one can be used instead, and the content of the array modified at will. Since the compiler recognizes and optimizes static final accesses, they don't experience the slowdown of non-final static variables. The results in Figure 3 show that indirection is much faster despite the extra memory reference on each access.

Unfortunately, the slowdown of static accesses does not appear to affect the AMR Titanium code significantly. Static variables are only used for timing purposes, and

replacing them with indirect accesses only marginally affected performance.

5.2 False Sharing

The other possible explanation for the Pthreads slowdown is false sharing. Since the Boehm-Weiser garbage collector is not thread-aware, it is possible that allocation requests from different threads are serviced with contiguous memory locations, resulting in false sharing. In order to determine the likelihood of this scenario, we used PAPI counters to measure cache misses for the slow pieces of code under both processes and Pthreads.

As Figures 3, 4, and 5 show, the slowdown in initialization time between processes and Pthreads is accompanied by a large increase in the number of L1 instruction and data cache misses, lending credence to the false sharing theory. However, in CFInterp, no increase in misses is detected for fewer than eight processors, though there is still a slowdown in running time. Coupled with the fact that CFInterp allocates no new data structures, this seems to indicate that false sharing may not be the culprit in CFInterp. We have not yet determined, however, what an alternative cause could be.

5.3 Attempted Optimizations

Besides replacing static accesses with indirect accesses, we attempted a few more optimizations in order to reduce the slowdown experienced by Pthreads. Specifically, we attempted to combat false sharing using two schemes: region-based memory allocation and padding allocations.

The Titanium language allows a user to explicitly manage memory using regions. Since objects in regions aren't allocated by the garbage collector and distinct regions almost never share cache lines, false sharing can be eliminated using regions. In the initialization phase, we added region-based allocation of domains for both the slow code in `QuadCFInterp.ti` and `LevelFluxRegister.ti`, deleting the regions on exit. This resulted in large gains and good scaling for the former as shown in Figure 6, but inconsistent results for the latter. Unfortunately, this optimization results in subsequent segmentation faults since some domains allocated in these regions need to persist after the initialization. Allowing the entire regions to persist is not an option, as it increases runtime many-fold due to memory leakage. We believe, though, that with proper knowledge of the AMR algorithm, such domains can be singled out and allocated elsewhere.

The second optimization scheme we tried was to modify the compiler to pad every memory allocation by a predetermined number of bytes in order to prevent multiple objects from sharing a cache line. Since a cache line on Seaborg is 128 bytes, we tried padding by 32, 64, and 128 bytes. However, as shown in Figure 7, none of these resulted in any improvement.

5.4 Work in Progress

We attempted a third optimization as well, involving the actual implementation of Titanium domains. At the current moment, general domains are represented by a list of rectangular domains. This is spatially inefficient since a list node must be allocated for each element, and may contribute to false sharing due to the small size of such nodes. We attempted to implement an array-based version of domains, but due to time constraints, were not able to get it to work as of this writing.

An additional optimization we considered, but did not pursue due to time constraints, is to optimize the representation of domains for difference operations. The three slow lines in the AMR initialization all perform this operation, so optimizing it at the cost of other operations may be worthwhile. One possibility is to introduce some ordering among rectangular domains, and force the list in a general domain to be sorted. This reduces the cost of a difference operation from quadratic to linear, though it increases the cost of a union. Unfortunately, it would be difficult to implement this in conjunction with the array-based optimization above.

6 Load Balancing

There are several viable strategies for allocating the regions among processors. We are not aware of the distribution algorithm used to create the sample AMR input files, so we have attempted to blindly evaluate the load balancing by gathering empirical evidence. We consider two simple strategies for allocating the boxes:

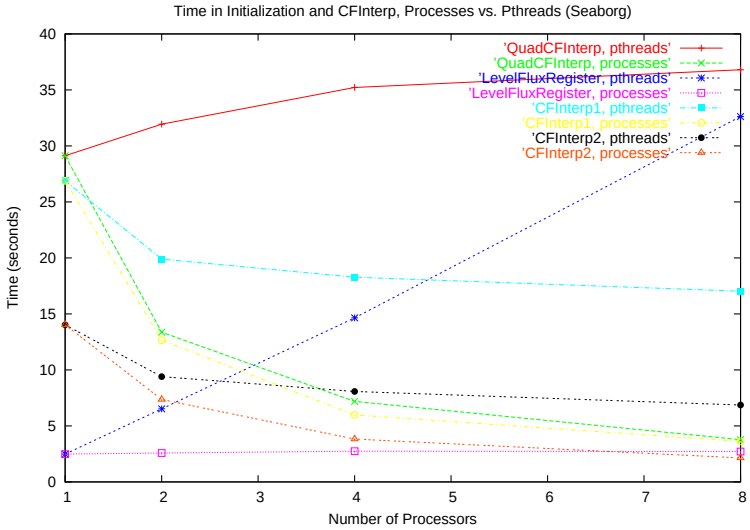


Figure 3: Time in initialization and CFInterp using processes and Pthreads.

1. [*Minimize Communication.*] Recall that when a coarse grid contains finer grids, there is an interpolation step in both directions of the V-cycle. If the fine grid(s) and coarse grid happen to reside on different processors, there is communication. To minimize communication, the distribution algorithm may be tempted to favor allocation either spatially-close regions, or regions with the containment relation to a single processor. This of course must strike a healthy balance with load-balancing. In the extreme, all communication could be eliminated by putting all boxes on one processor. In the other extreme, each processor gets one box, and every interpolation requires communication.

2. [*Equal Work.*] Alternatively, ignoring the above observations, a distribution algorithm could simply try to allocate an equal workload to each processor; for instance, an equal number of points, or an equal volume of space in the 3-dimensional AMR cube.

   One simple way to measure the imbalance of work is to look at the time each processor spends waiting at the barrier. In particular, we are interested in the maximum of all waiting times among all processor at any given barrier. This represents the case for the processor that reached the barrier first, and hence waited the longest. Figure 8 shows a graph of the load balancing characteristics for 5 problem sizes on 5 processor configurations.

Figure 8 shows the percent of the total running time that was spent waiting on barriers. The graph shows that when the larger configurations are used, the time spent waiting at barriers can account for over one third of the total execution time [1]. This suggests that the biggest source of improvement could come from choosing an algorithm that favors equal distribution of work over less communication. As network speeds increase, this becomes increasingly more important.

[1] To verify that the implementation of barriers in Titanium was not the culprit, we constructed a microbenchmark. The time spent was negligible compared to the time in the AMR code.

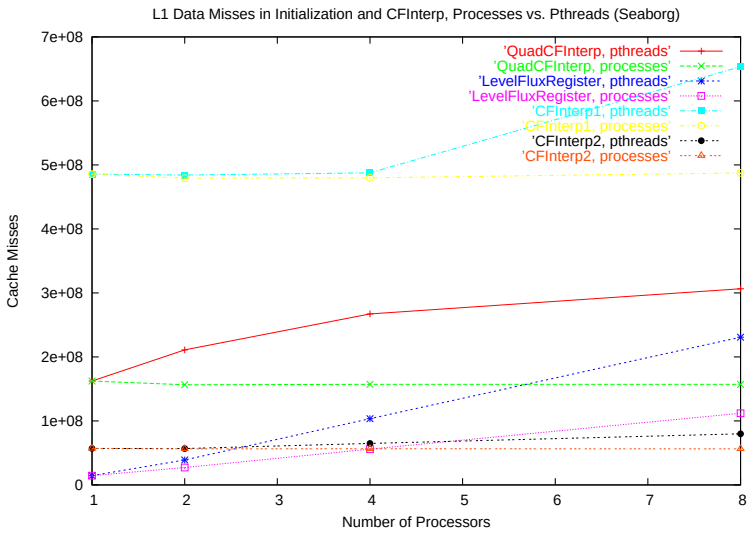


Figure 4: L1 instruction cache misses in initialization and CFInterp using processes and Pthreads.

7 Exchange

In the exchange method, elements in the intersection of two boxes are copied. Depending on the box assignments, the two boxes can be on two different processors. The original implementation of the exchange method uses two different methods for copying the elements in the intersection. If the two boxes live on the same processor, then the copy is done using a foreach loop to do an element by element copy. If the two boxes live on different processors, then a call to array copy is used.

Ideally, only the array copy implementation is needed. The foreach version can be viewed as a user implementation of the array copy method. The reason for having two implementations is performance. The copy using array copy is about 50% slower than the foreach version. This is due to the different number of intersection operations in the two versions. The array copy version uses two intersection operations, where the foreach version uses one. For small intersections, the cost is dominated by the intersection operations.

7.1 Intersection Size Distribution

Figure 9 shows the distribution of the intersection sizes for our input. About 20% of the intersections have size one. More than 50% of the intersections have less than 10 elements.

7.2 Optimizations

Due to the large number of small intersections, we want to reduce the number of intersection operations. 100% of the single element intersections are due to the intersection of the corners for two boxes. The corners are actually not used for computation, so those copies are not necessary. We reduce the amount of time spent in exchange from 74 seconds to 33 seconds by checking for this case.

The boxes are static during the execution of the program. Therefore, the intersections do not change between iterations. We decided to amortize the cost of the intersection operations by caching the results. This optimization speeds up the exchange method by 10% on top of the previous optimization.

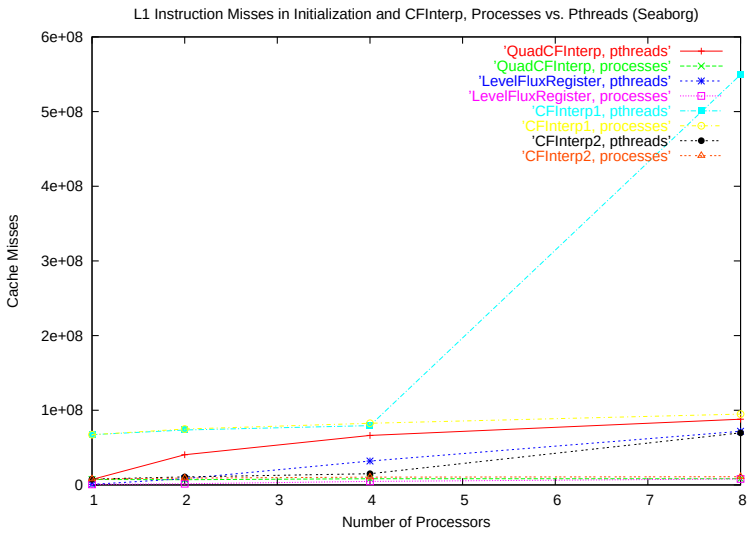


Figure 5: L1 data cache misses in initialization and CFInterp using processes and Pthreads.

7.3 Language Comparison

The exchange method code size for the Titanium version and the C++/MPI version differs drastically. In terms of lines of code, the Titanium version uses 47 lines of code, and the C++/MPI version uses 337 lines of code. In the C++/MPI version, only 9 out of the 337 lines of code are not MPI related. About 50% of the MPI related code is for buffer management. There is no explicit buffer management in the Titanium version. Communication is implicit through reads, writes, and array copies.

8 Performance Results

We run our experiments on Seaborg and Seberg. Seaborg has 375 MHz POWER3 processors. Seberg is an uniprocessor machine with a 2.8 GHz Pentium 4 processor. The size of the input grids is described in Table 4.

We have three versions of the code. There are two versions of the Titanium code, one for before the project, and one for after the project. The before version is named Titanium 1, and the after version is named Titanium 2. We also have data for the C++/MPI Chombo code. First, we compare the sequential performance in Table 5. The optimizations done in the project speed up the Titanium code by around 23% on both machines. On Seaborg, it is 2.25 times faster than the C++/MPI code in the sequential case. On Seberg, the optimized Titanium code is 7% slower than the C++/MPI code. We have not tried applying the optimizations we did for the Titanium code on the C++/MPI code, but we suspect that there is room for improvement in the C++/MPI code. The large gap in performance between the Titanium code and C++/MPI code may be narrowed in the coming days. The Chombo group in LBNL has found a performance bug for the code on Seaborg.

Table 6 compares the performance of the two Titanium codes on the smp backend on Seaborg. It uses a single node with different number of processors.

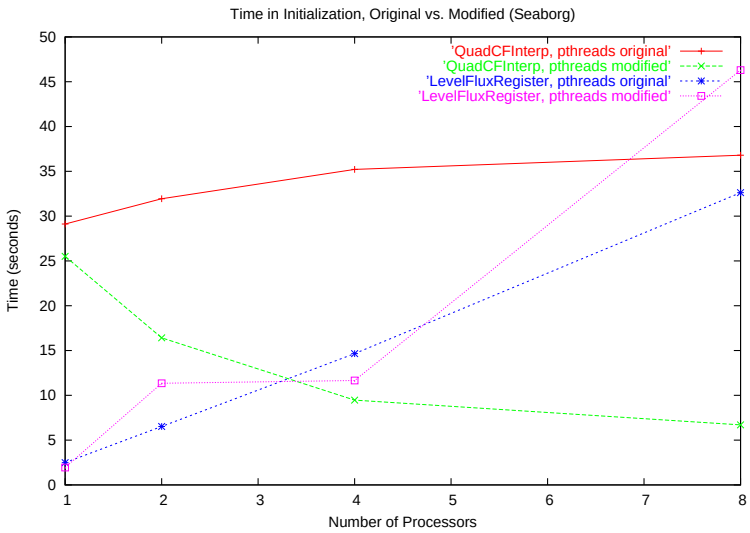


Figure 6: Time in initialization using regions.

Level	# of boxes	# of points
0	1	32768
1	106	279552
2	1449	2944512

Table 4: Input grid size.

	Seaborg	Seberg
Titanium 1	208 secs	89 secs
Titanium 2	162 secs	68 secs
C++ Chombo	366 secs	62 secs

Table 5: AMR sequential running time.

	1 proc	2 procs	4 procs	8 procs	16 procs
Titanium 1	222 secs	131 secs	83 secs	81 secs	86 secs
Titanium 2	169 secs	103 secs	70 secs	65 secs	78 secs

Table 6: AMR Titanium parallel running time (smp backend).

9 Conclusions

We have presented analysis and several optimizations for a Titanium implementation of AMR. Our most effective optimization was the modification to the exchange method as described in Section 7.3. Ultimately, it resulted

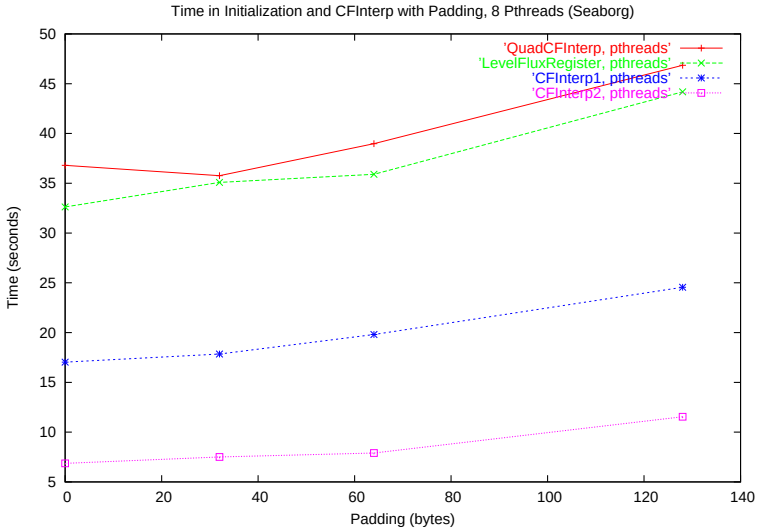


Figure 7: Time in initialization and CFInterp using padding.

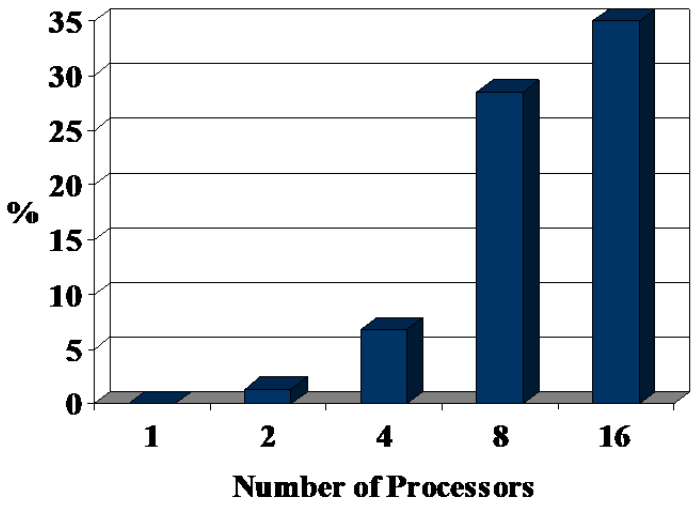


Figure 8: Load Imbalance: Percent of Time Spent Waiting

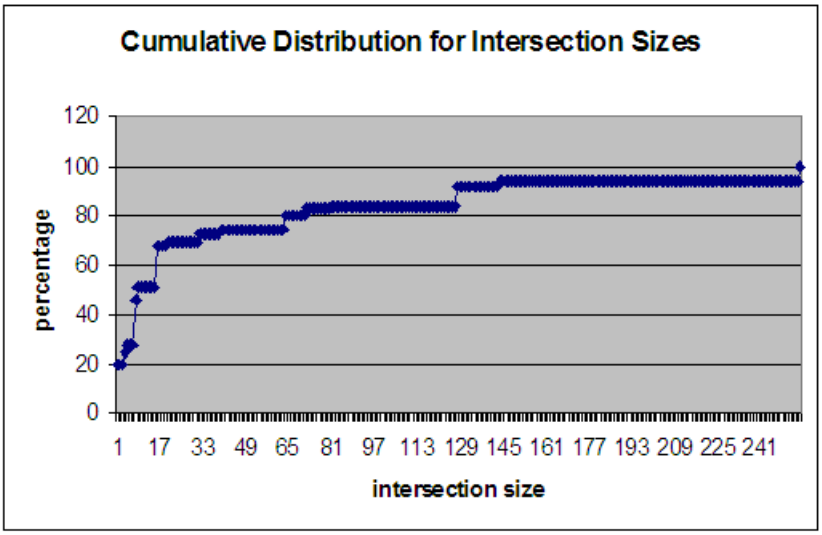


Figure 9: Cumulative distribution of intersection sizes in exchange.

in a speedup of 23%. The Titanium implementation is either comparable, or faster than the Chombo MPI/C++ implementation depending on the platform.

We also found that load balancing was playing a significant role in limiting the program from achieving a higher peak performance. Additionally, we performed several promising optimizations related to false sharing and presented the performance results.

References

[1] Chombo design document.

[2] Paul Hilfinger. Titanium language reference manual. `http://titanium.cs.berkeley.edu/doc/lang-ref.ps`.

[3] Dan Quinlan. Amr++: Object-oriented design for adaptive mesh refinement, 1994.

[4] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, , and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13), September-November 1998.