Title
Enforcing Textual Alignment of Collectives Using Dynamic Checks

ISBN
978-3-642-13373-2

Authors
Kamil, Amir
Yelick, Katherine

Publication Date
2010

DOI
10.1007/978-3-642-13374-9_25

Peer reviewed

Enforcing Textual Alignment of Collectives Using Dynamic Checks *

Amir Kamil Katherine Yelick

Computer Science Division, University of California, Berkeley
{kamil,yelick}@cs.berkeley.edu

Abstract. Many parallel programs are written in a single-program, multiple-data (SPMD) style, in which synchronization is provided using collective operations that all threads execute simultaneously. If these operations are not properly aligned on all threads, deadlock can occur, and many compiler analyses and optimizations that depend on proper alignment fail. In this paper, we discuss the flaws in the Titanium language's type system for enforcing textual alignment of collectives. We then present a system that uses runtime checks to ensure alignment for two definitions of textual alignment. The system instruments the code to keep track of alignment in each thread and then checks that alignment matches prior to performing a collective operation. We have implemented the system in the Titanium compiler, verifying that it catches alignment errors. We tested its performance on multiple application programs, demonstrating that the checks have no appreciable impact on execution time.

1 Introduction

With the growing body of parallel programs for both multicore and high performance cluster systems, there is a need for good program analysis techniques to detect programming errors and enable optimizing transformations. Critical to both error detection and optimization is an analysis of the synchronization constructs of a program. In this paper, we consider the problem of alignment of barrier synchronization operations and other operations that all threads perform collectively, such as broadcasts, which implicitly result in synchronization.

Prior work on the Titanium language [20] has shown the value of synchronization analysis, that it enables communication optimizations, race detection, and enforcement of memory consistency models [12, 13]. We present an overview of Titanium's synchronization model, which requires that all threads can be proven statically to reach the same textual instance of each collective [2]. This model is very powerful, but our experience indicates that programmers find the static checking overly restrictive. We propose a dynamic version of the analysis that relaxes the language definition but still permits the benefits of static checking: the compiler may assume barriers are textually aligned,

* This work was supported in part by the Department of Energy under DE-FC03-01ER25509, FDDE-FC02-07ER25799, and Lawrence Berkeley National Laboratory Contract DE-AC02-05CH11231, by the National Science Foundation under CNS-0325873 and OCI-0749190, by the California State MICRO Program, by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), and by gifts from Sun Microsystems. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

but final checks are made at runtime. We implemented two versions of a fully dynamic analysis and show that while individual collective operations are more expensive due to the runtime checking, the application overhead is negligible. Since the dynamic model does not depend on the strong typing properties of Titanium, it is also applicable to the broader class of PGAS languages that rely on a SPMD execution model, including UPC [5] and Co-Array Fortran [16].

2 Background

Titanium [20] is a dialect of Java but does not use the Java Virtual Machine model. Instead, the end target is assembly code. For portability, Titanium is first translated into C and then compiled into an executable. In addition to generating C code to run on each processor, the compiler generates calls to a runtime layer based on GASNet [4], a lightweight communication layer that exploits hardware support for direct remote reads and writes when possible. Titanium runs on a wide range of platforms including uniprocessors, shared memory machines, distributed-memory clusters of uniprocessors or SMPs (CLUMPS), and a number of specific supercomputer architectures (Cray X1, Cray T3E, SGI Altix, IBM SP, Origin 2000, and NEC SX6). Instead of having dynamically created threads as in Java, Titanium is a *single program, multiple data* (SPMD) language, so all threads execute the same code image.

2.1 Collective Operations

Many scientific applications are written in a bulk-synchronous style that alternates between communication and computation phases, or between different phases of physical simulations such as the ocean and atmospheric models in a climate simulation. These applications frequently require all threads to synchronize and communicate together. Like other SPMD languages, Titanium provides *collective operations* to support this. The three primitive collective operations in Titanium are barriers, broadcasts, and exchanges. A *barrier* forces threads to wait until all threads have reached it. A *broadcast* is a one-to-all communication construct that sends a value from one thread to the others. An *exchange* is an all-to-all communication construct that copies one value from each thread to all threads.

Collective operations introduce the possibility of deadlock if not all threads execute the same sequence of collectives. The collectives are *aligned* if all threads do execute the same sequence.

Most SPMD languages do not attempt to guarantee alignment of collectives. Some languages such as UPC have *named* collectives. These collectives take an integer value as an argument. When the collective executes, it compares the value on all threads and generates an error if they differ. However, different collective operations in a program can have the same value under this scheme. Even if each collective in a program has its own unique value, as soon as the collective is wrapped inside a function, the alignment scheme can be defeated. For example, a call to the following function acts like an unnamed barrier:

```
void barrier2() {
  upc_barrier 315415431;
}
```

More complicated and less malicious examples of this can occur in practice. A further flaw with named collectives is that they can result in late error messages: the actual program statement that causes misalignment can be far from the affected collective and is not detected or reported to the user.

Aiken and Gay introduced the concept of *structural correctness* to enforce alignment of collectives and developed a static analysis that determines whether or not a program is structurally correct [2, 9]. The following code is not structurally correct:

```
if (Ti.thisProc() % 2 == 0)
  Ti.barrier(); // even ID threads
else
  ; // odd ID threads
```

Titanium provides a stronger guarantee of *textually aligned collectives*: not only do all threads execute the same number of collectives, they also execute the same *textual* sequence of collectives. Thus, both the above structurally incorrect code and the following structurally correct code are erroneous in Titanium:

```
if (Ti.thisProc() % 2 == 0)
  Ti.barrier(); // even ID threads
else
  Ti.barrier(); // odd ID threads
```

The fact that Titanium collectives are textually aligned not only guarantees deadlock freedom but is also essential to concurrency analysis: not only does it guarantee that code before and after each collective cannot run concurrently, it also guarantees that code immediately following two different collectives cannot execute simultaneously [12, 13].

Titanium currently relies on a static type system to ensure textual alignment of collectives. We discuss this type system in §3.

2.2 Related Work

In addition to Aiken and Gay's work on structural correctness, many others have addressed the problem of collective alignment.

Zhang and Duesterwald developed a static analysis for matching textually unaligned barriers in MPI [21]. The analysis is available as part of the Eclipse Parallel Tools Platform [1]. They later extended their analysis in collaboration with Gao to shared memory OpenMP programs and built a concurrency analysis on top of it [22]. Siegel and Avrunin applied model checking to MPI programs [18]. Their system detects deadlock in the presence of MPI barriers.

Jeremiassen and Eggers developed a static analysis for barrier synchronization for SPMD programs with non-textual barriers that divides a program into non-concurrent

phases [10]. This analysis provides a conservative estimate of which barriers can conflict: two barriers can only conflict if they may both be executed at the boundary of the same phase. Other work has also been done in the area of concurrency analysis [14, 15], though like Jeremiassen and Eggers, the authors don't directly apply it to detecting alignment errors.

A lot of work has also been done in the area of barrier optimization [6, 17, 19], which requires reasoning about the execution of barriers. However, work in this area generally either assumes aligned barriers or is not concerned with detecting synchronization errors arising from the misaligned barriers.

The main drawback to static analysis is imprecision: it may be unable to conclude that a collective is properly aligned even if it is. While in practice, results from static analysis appear to be precise enough to be useful for other analyses and optimizations, the existence of false positives makes it less than ideal for enforcing semantic restrictions, as it would report nonexistent errors to the programmer.

3 The `single` Type System

In order for collectives to be textually aligned, all expressions that control the execution of collectives must evaluate to coherent values on all threads. In Titanium, the *`single` type system* ensures that this is the case. A value is *`single`* if it is coherent across all threads. The entire set of rules for what values are `single` is described in the Titanium language reference [20] and is fairly complicated, so we describe only a subset of the rules here.

3.1 `single` Values

For primitive types, the coherency rules are straightforward. Compile-time and runtime constants (such as the number of executing threads) are `single`, as well as expressions composed entirely of `single` values. Variables are `single` if they are annotated as such by the programmer. Such variables can only by assigned with `single` values.

The rules for method calls are more complex. In order for the result of a method call to be `single`, its return type must be declared as `single`, the object being dispatched on must be `single` for an instance method, and all parameters declared as `single` must be passed `single` arguments. These rules are illustrated below:

```
class Foo {
  int single bar(int single x, int y) { ... }
  static void baz() {
    Foo single a = ...;
    Foo b = ...;
    int single i = 1;
    int j = Ti.thisProc();
    a.bar(i, j); // return is single
    b.bar(i, j); // return is not single since b is not
    a.bar(j, i); // return is not single since x = j is not
```

```
  }
}
```

An allocation results in a `single` object if the constructor call obeys the rules for method calls above. A field dereference is `single` if the referenced object is `single` and the field is declared as `single`.

3.2 Control Flow Restrictions

All control flow decisions that affect the execution of statements with global effects must only depend on `single` expressions. In general, a statement has *global effects* if it or any of its substatements is one of the primitive collective operations described in §2.1, a method call that a programmer has declared as global by qualifying it with the `sglobal` keyword[1], or assigns to certain locations that are declared as `single`.

All branches, loops, and method calls that have global effects can only be controlled by `single` expressions. Exceptions have more complicated restrictions, and the interested reader can refer to the Titanium language reference for the associated rules.

3.3 Type System Flaws

As hinted at above, the `single` type system rules are complicated. Feedback from Titanium users indicates that while they appreciate the fact that the type system prevents deadlock, the error messages can be confusing at times due to the conservative nature of the analysis.

The type system requires the programmer to annotate many variables with the `single` keyword. Since a `single` variable can only be assigned an expression composed of other `single` variables, it may be necessary to propagate these annotations throughout a program. This can be quite burdensome, especially for quick prototyping of small pieces of code.

There are additional flaws in the type system, such as its handling of arrays and casts to `single` [11]. Most importantly, we have not yet found a clean way to extend the type system to allow collectives over a subset of all the threads.

3.4 Subset Collectives

The Titanium language reference contains a proposal for a `partition` construct that divides the set of threads into non-overlapping teams. Collective operations affect only members of the team that execute them. In the below code, only those threads whose IDs are less than 3 will call `setX(1)`.

```
static single void bar() {
  partition {
  Ti.thisProc() < 3 => setX(1);
```

[1] In the current Titanium language specification, the `sglobal` keyword has been deprecated in favor of the `single` keyword. In this paper, however, we will use `sglobal` to prevent confusion with non-global methods that have a `single` return or `single` arguments.

```
    }
    if (x == 0)
      Ti.barrier(); // misaligned barrier
  }
  static int single x = 0;
  static single void setX(int single y) { x = y; }
```

Since `single` is used to ensure alignment of collective operations, it only implies coherence across all threads of a given team. This can cause problems if a `single` variable is updated within a `partition` statement. In the above code, a subset of the threads modify the `single` variable `x`. Since collectives are over a team, the code is correctly typed. However, it will deadlock, since the barrier after the `partition` is executed by all the threads, which now have incoherent values of `x`.

4 Dynamic Alignment

Given the flaws of the `single` type system described in §3.3, we present an alternative, dynamic scheme for enforcing textual alignment of collectives.

4.1 Alignment Rules

The basic conditions that guarantee textual alignment of collectives are as follows[2]:

- If any branch of a conditional has global effects, then all threads must take the same branch.
- If the body/test of a loop has global effects, then all threads must execute the same number of iterations.
- If a method call has global effects, then the dynamic dispatch target of the call must be the same on all threads.
- The source thread in a broadcast expression must evaluate to the same value in each thread.

All four conditions above are enforced by the `single` type system.

A strict alignment scheme enforces the above conditions with respect to a similar definition of *has global effects* as the `single` type system: a statement has global effects if it or any of its substatements is a primitive collective operation or calls a method declared as `sglobal`. A weaker alignment scheme results if a statement has global effects only if it or any of its substatements *actually executes a primitive collective operation at runtime*. In particular, the following code is legal under weak alignment but prohibited under strict alignment:

```
if (Ti.thisProc() % 2 == 0) // even threads
  if (Ti.thisProc() % 2 == 1) // odd threads
    Ti.barrier(); // never reachable
```

[2] We omit discussion of exceptions here, as the rules are essentially the same as in the `single` type system.

Under weak alignment, the code above never executes the barrier, so it does not have global effects and not all threads must take the same branch of the outer conditional. Under strict alignment, however, the *then* branch does have global effects since one of its substatements is a barrier, so all threads must take the branch.

Note that the strict alignment rules are static: it is possible to determine which statements have global effects at compile-time. The weak alignment rules, on the other hand, are partially dynamic. For some statements, it can be statically determined that they never execute primitive collectives. For others, however, it can only be determined at runtime whether or not they do so. As a result, we believe that strict alignment is preferable both for compiler analysis purposes and for programmer reasoning.

4.2 Dynamic Enforcement

The alignment rules are enforced dynamically by tracking those conditionals, loops, and method calls that have global effects. (For the purposes of this section, global effects are according to the strict definition above.). The Titanium compiler has an inference system that statically determines which statements have global effects.

At program startup, each thread creates an empty list that records its execution history. In addition, a hash of this list is maintained, initially set to some value h_0. The following operations update the list, with the hash updated accordingly:

- On a non-`static` dispatch to a method that has global effects, an entry is added to the list with the method that is the dynamic dispatch target.
- On a branch of a conditional that has global effects, an entry is added recording the branch taken.
- On each iteration of a loop that has global effects, an entry is added recording that a loop iteration occurred.
- On reaching a broadcast operation, an entry is added with the value of the source thread.

When performing a primitive collective, the hashes for all threads are first compared. This can be done by using a comparison tree, or simply by broadcasting the hash value from a single thread and comparing it to the value on each other thread. Our implementation currently uses the latter. If the hash values match, the collective is executed. If any two hashes differ, however, then execution halts, and the corresponding histories are used to generate an appropriate error message. For performance reasons, the execution history list can be eliminated or reduced in size, at the cost of poorer error messages.

The above procedure is sufficient for enforcing strict alignment. Weak alignment, on the other hand, only guarantees alignment of statements that execute primitive collectives at runtime. Thus, if a statement never executes a primitive collective, the execution history and hash must be restored to their previous values once the statement completes. This necessitates saving the old history and hash state before making any of the above changes.

Figure 1 illustrates the execution of the following code under strict and weak alignment:

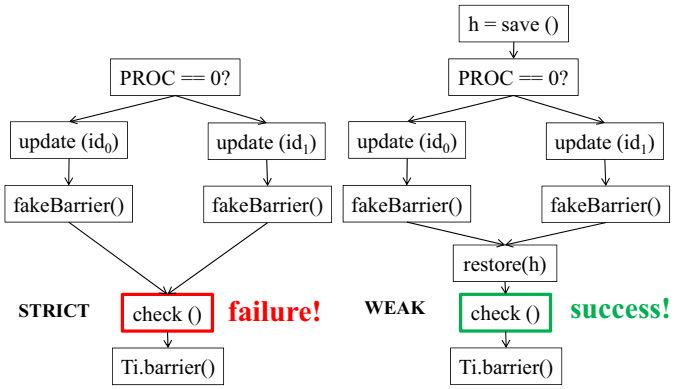

Fig. 1. Difference in execution between strict and weak alignment.

```
if (Ti.thisProc() == 0) fakeBarrier();
else fakeBarrier();
Ti.barrier();
```

Here, `fakeBarrier` is a method declared as `sglobal` but that does not execute any primitive collective operations. As such, the code should fail under strict alignment but succeed under weak alignment.

Optimizations It is possible to reduce the number of history updates and checks if they can be proven redundant. If two history updates always occur in sequence with no intervening collectives, then they can be combined into a single update. This may occur in nested conditionals or conditionals inside loops. Similarly, two history checks can be redundant if no updates occur between them.

Another optimization is a hybrid static/dynamic analysis that would apply a static analysis to determine which branches, loops, and method calls can be proven to depend only on `single` values and then eliminate their associated updates. This would also make it much more likely for consecutive collectives not to have any updates between them, allowing their associated history checks to be removed.

Program Coverage As is usually the case with dynamic analysis, the enforcement scheme above does not provide full program coverage, which is a drawback compared to the static type system. In particular, it does not check alignment of code that is not reached at runtime, such as the following:

```
if (<some rare condition>)
  if (Ti.thisProc() == 0)
    Ti.barrier();
```

An error is only generated if the rare condition is taken. Similarly, if the rare condition was replaced by an expression dependent on the number of threads, such as

`Ti.numProcs() > 100`, then an error would only occur if the program was run with the required number of threads to trigger the condition.

Weak alignment provides somewhat less coverage than strict alignment. Consider the above code with the two conditions switched. Now, strict alignment would generate an error since the threads are not aligned with respect to the outer conditional, which has global effects. Weak alignment, on the other hand, will only find an error if the rare condition is met, since no primitive collective operation is executed otherwise.

4.3 Subset Collectives

The dynamic enforcement scheme can easily be extended to the `partition` construct described in §3.4. At the start of such a statement, the current execution history and hash need to be saved. Within a `partition`, the history and hash are updated as anywhere else. When performing a team collective, the hash is compared only among the threads in the associated team, guaranteeing that these threads are aligned. At the conclusion of a `partition`, the saved history and hash are restored, so that the differing team alignments within the `partition` do not affect code outside the `partition`.

4.4 Implementation

We have implemented the two dynamic enforcement schemes in the Titanium compiler. The compiler instruments each program to perform the required tracking and checking[3]. We do not apply the optimizations described in §4.2, since our experimental results in §5.2 show them to be unnecessary. We also do not yet support subset collectives, as GASNet does not yet officially allow them.

For both strict and weak alignment, the compiler provides a default mode where only a hash is kept corresponding to execution history as well as a debugging mode that maintains the execution history list. The default mode requires less memory and fewer operations at runtime than the debugging mode, so it could potentially be more efficient. The debugging mode only stores execution history between successive primitive collective operations, since successful completion of a collective implies that it and all statements preceding it are properly aligned.

The Titanium compiler provides an escape hatch for when dynamic error checking adversely affects performance. Users generally switch to a less-safe, higher-performance mode that elides error checking such as `null` pointer and array bounds checks for production runs, under the assumption that such errors have already been caught while debugging a program. This escape hatch can also be used if dynamic alignment checking proves to be too expensive, as the compiler can remove those checks as well.

5 Evaluation

We have verified on many different test cases that the dynamic enforcement system does detect alignment errors in practice without requiring any programmer annotations.

[3] Note that for weak alignment, we could instead examine the program stack on each thread. However, this would require a far more complicated implementation, as stack layout depends on the target machine and the C compiler.

On the program corresponding to Figure 1, the following error is produced under strict alignment in debugging mode in addition to the usual Java-like stack trace:

```
ti.lang.Alignment.AlignmentError: collective alignment
 failed on processor 1 at foo.java:9:8
last location: else branch at foo.java:5:12
last location on processor 0: then branch at foo.java:5:12
previous location: none
```

The error message directs the user to the exact location that caused alignment to fail, instead of just providing the location of the misaligned collective itself.

Since Titanium currently enforces alignment statically, no Titanium application has alignment errors, and we could not test its effectiveness on real-world programs. However, we were able to determine the performance cost of dynamic checking on two platforms: an eight-core (two-processor, four-core) Intel Xeon E5435 shared memory multiprocessor (SMP) and a cluster composed of dual-processor 2.2GHz Opteron nodes with an InfiniBand interconnect.

Five program versions were compared:

- `static`: no dynamic checking
- `strict`: strict alignment scheme, default mode with no execution history list
- `strict/debug`: strict alignment scheme, debugging mode with execution history list
- `weak`: weak alignment scheme, default mode with no execution history list
- `weak/debug`: weak alignment scheme, debugging mode with execution history list

5.1 Collective Performance

We first tested the performance of each of the primitive collectives by repeatedly invoking them inside a loop. For the dynamic alignment schemes, a single loop iteration includes an update to the execution hash (and list for the debugging modes), and for the broadcast test, an additional execution hash/list update for the source thread of the broadcast. Each loop iteration also includes the hash comparison code associated with a collective operation, which consists of a broadcast, comparison, and conditional.

Figure 2 shows the relative loop iteration time for the broadcast, barrier, and exchange tests on the SMP machine for up to 8 processors. The broadcast is of a single 32-bit integer, and the exchange is of 32-bit integers among all threads. On average, barriers in the dynamic schemes take about 2.7 times as long as the static version, broadcasts take 2.5 times as long, and exchanges 70% longer. The dynamic debugging versions consistently were slower than the default dynamic versions by an average of about 20%.

Figure 3 shows the loop iteration times for each collective in the static case on the cluster machine for up to 32 processors, and Figure 4 shows the dynamic times relative to the static time. The broadcast and exchange are as in the SMP case. The jump in barrier time at four processors is due to it being the fewest number of processors to require internode communication. On average, barriers in the dynamic schemes take

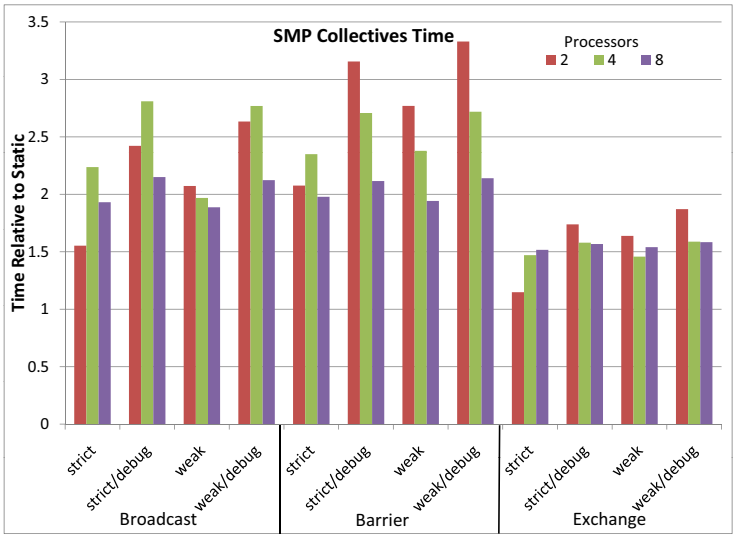

Fig. 2. Relative collective performance on the SMP machine.

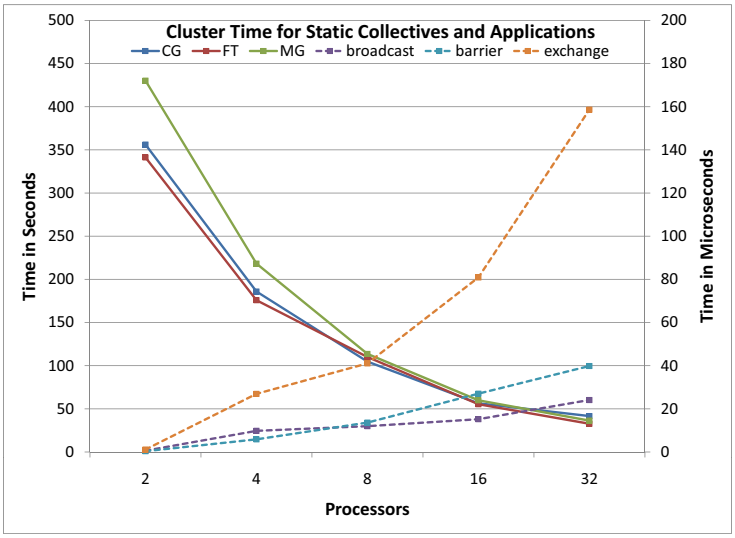

Fig. 3. Collective and application performance on the cluster machine in the static case. Application times are plotted with respect to the left vertical axis and collective times with respect to the right.

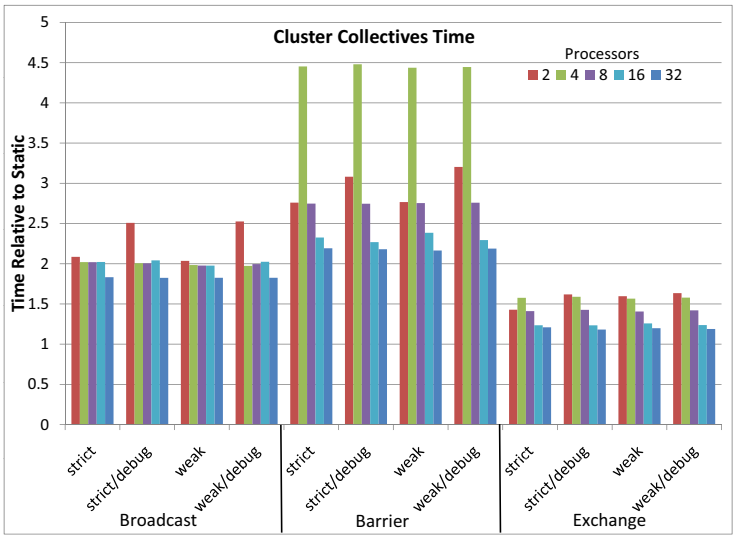

Fig. 4. Relative collective performance on the cluster machine.

about three times as long as the static version, broadcasts take twice as long, and exchanges 40% longer. There is little discernible difference between the various dynamic versions.

On both machines, the overhead of dynamic checking decreases for each collective operation as the number of processors increases. In particular, the cost of an unchecked broadcast trends to about half that of a barrier on the cluster machine and becomes negligible compared to the cost of an exchange. Since checked operations include an extra broadcast, we expect that for a large number of processors, broadcasts would take twice as long with dynamic checking compared to static, barriers would take 1.5 times as long, and exchanges would take about the same amount of time.

5.2 Application Performance

We also tested three of the NAS Parallel Benchmarks [3] in Titanium [8, 7]: conjugate gradient (CG), Fourier transform (FT), and multigrid (MG). Due to memory constraints, the three benchmarks used class B, A, and B sized problems, respectively, though the parameters were tweaked to increase running time.

In general, parallel applications do not perform collectives in their inner, compute-intensive loops, nor do such loops tend to have global effects. Thus, even though collectives are slower under dynamic alignment and tracking operations have some additional cost, we expect them to be executed rarely and do not expect them to drastically affect application performance. Figure 3 shows the running time for the NAS Parallel Benchmarks on the cluster machine in the static case for up to 32 processors. Figure 5 shows the relative dynamic times, demonstrating that for these applications, the dynamic checks have no effect on performance. The same results also occurred on the SMP, though the graphs have been omitted for brevity.

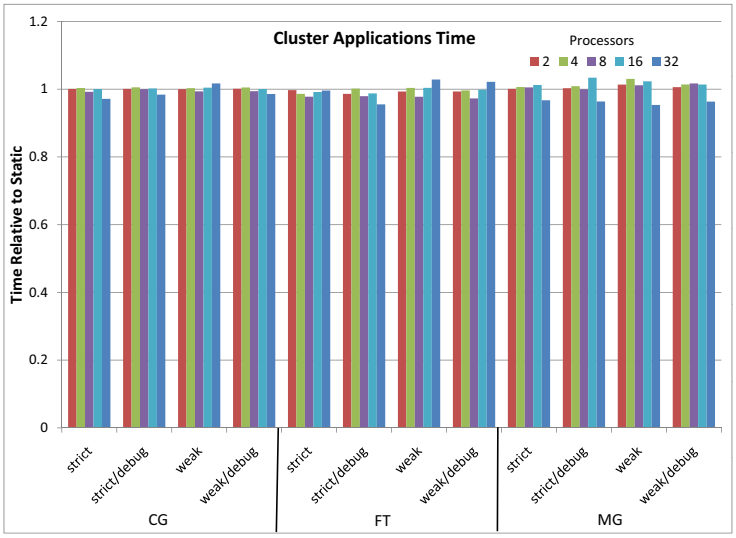

Fig. 5. Relative application performance on the cluster machine.

Analysis In order to understand why dynamic checking does not appear to significantly affect application performance, we ran experiments to determine how much impact it should have. In Table 1(a), we report the amount of time it takes for each alignment operation on the two machines, including the time to update the alignment hash, the time to update the history and hash in debugging mode, the time to perform a history save and restore pair for weak alignment, and the time to perform a hash check. The latter varies depending on the number of processors, so we report the maximum time on all processor counts we tested.

Table 1(b) shows the number of times each operation occurs in each benchmark. Using these numbers and the times from Table 1(a), we conservatively estimate the maximum effect on running time for each machine. The worst case is 1.37 seconds for the multigrid application on the cluster, which is less than 5% of total running time on 32 processors. The results in Figure 5 show that the actual effect is even less than this.

Table 1. Time for alignment update and check operations on each machine, and number of operations and calculated overhead for each benchmark.

(a)

Time	SMP	Cluster
Update	4.0 ns	12.8 ns
Debug Update	31.4 ns	152.3 ns
Save/Restore	10.3 ns	87.2 ns
Max Check	1.6 μs	47.5 μs

(b)

	Operation Count			Max Total Time	
	Updates	Saves/Restores	Checks	SMP	Cluster
CG	1844	924	2729	4.4 ms	0.13 s
FT	1835	1216	1218	2.0 ms	0.05 s
MG	100530	69248	28320	48.9 ms	1.37 s

In general, most applications try to avoid collectives since they limit scalability. As a result, they would not be affected much by the overhead of dynamic checking at each collective. Applications that spend a significant fraction of their time in collectives can expect this portion of their execution time to as much as double for large numbers of processors, as we argued in §5.1. If this cost is too high, users can turn of checking as noted in §4.4. In addition, the optimizations described in §4.2 should significantly reduce the performance impact of dynamic checking.

We also determined how much space the execution history list uses in debugging mode. Since entries are cleared from the list at each collective, the number of entries in the list when performing a collective is equal to the number of control flow decisions that affect execution of the collective since the previous collective executed. We expect this number to be relatively small and the execution history to use little space as a result. Our tests showed that the space used on each thread was 0.8 KB. 0.3 KB, and 90 KB for CG, FT, and MG, respectively, matching our expectations.

6 Conclusion

We presented Titanium's synchronization model for barriers and other collective operations, which requires that all threads must reach the same textual instance of each collective for a program to be correct. We then described a system of dynamic checks for enforcing proper alignment of collectives. The system has two variations, a strict version that matches the previous static type system in what must be aligned and a weak version that only requires primitive collective operations that get executed at runtime to be aligned. The dynamic system is more flexible and intuitive than the static one, requiring no programmer annotations while permitting the use of collectives on thread teams.

Although the dynamic checks used for the runtime analysis result in collective operations that are up to three times slower than without the checks, we showed that the overhead is negligible in three application benchmarks on both a shared memory multiprocessor and a cluster machine. Our results indicate that dynamic checking is a viable replacement for the current static type system in Titanium and the unchecked synchronization in other languages, enabling analyses and optimizations that depend on proper alignment of collectives.

References

1. Eclipse Parallel Tools Platform. `http://www.eclipse.org/ptp/`.
2. A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.
3. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
4. D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, November 2002.

5. W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
6. A. Darte and R. Schreiber. A linear-time algorithm for optimal barrier placement. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 26–35, New York, NY, USA, 2005. ACM.
7. K. Datta. The NAS Parallel Benchmarks in Titanium. Master's thesis, University of California, Berkeley, December 2005.
8. 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)*, 2005.
9. D. Gay. *Barrier Inference*. PhD thesis, University of California, Berkeley, May 1998.
10. T. Jeremiassen and S. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, August 1994.
11. A. Kamil. Problems with the titanium type system for alignment of collectives, February 2006. `http://www.cs.berkeley.edu/~kamil/titanium/doc/single.pdf`.
12. A. Kamil, J. Su., and K. Yelick. Making sequential consistency practical in Titanium. In *Supercomputing 2005*, November 2005.
13. A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.
14. A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. 1996.
15. Y. Lin. Static Nonconcurrency Analysis of OpenMP Programs. In *First International Workshp on OpenMP (IWOMP 2005)*, June 2005.
16. R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.
17. M. O'Boyle and E. Stohr. Compile time barrier synchronization minimization. *Parallel and Distributed Systems, IEEE Transactions on*, 13(6):529–543, Jun 2002.
18. S. F. Siegel and G. S. Avrunin. Modeling wildcard-free MPI programs for verification. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 95–106, New York, NY, USA, 2005. ACM.
19. C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. *SIGPLAN Not.*, 30(8):144–155, 1995.
20. 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 *Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.
21. Y. Zhang and E. Duesterwald. Barrier matching for programs with textually unaligned barriers. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 194–204, New York, NY, USA, 2007. ACM.
22. Y. Zhang, E. Duesterwald, and G. R. Gao. Concurrency analysis for shared memory programs with textually unaligned barriers. In *Languages and Compilers for Parallel Computing, 20th International Workshop, LCPC 2007*, volume 5234 of *Lecture Notes in Computer Science*, pages 95–109, 2008.