Hierarchical Additions to the SPMD Programming Model

Abstract

Large-scale parallel machines are programmed mainly with the single program, multiple data (SPMD) model of parallelism. This model has advantages of scalability and simplicity, combining independent threads of execution with global collective communication and synchronization operations. However, the model does not fit well with divide-and-conquer parallelism or hierarchical machines that mix shared and distributed memory. In this paper, we define a hierarchical team mechanism that retains the performance and analysis advantages of SPMD parallelism while supporting hierarchical algorithms and machines. We demonstrate how to ensure alignment of collective operations on teams, eliminating a class of deadlocks. We present application case studies showing that the team mechanism is both elegant and powerful, enabling users to exploit the hardware features at different levels of a hierarchical machine and resulting in significant performance gains.

1. Introduction

Parallel languages use either dynamic or static models of parallelism, with dynamic threading models dominating small-scale shared memory programming and the static single program, multiple data (SPMD) model dominating large scale distributed memory machines. A SPMD program is launched with a fixed number of threads, typically one per core, that execute throughout the program. The SPMD model dominates programming at scale because it encourages "parallel thinking" throughout the program execution, exposing the actual degree of available parallelism. It also naturally leads to good locality and can be implemented by simple, low-overhead runtime systems. The model of communication between threads is orthogonal to choice of control, and both message passing models like MPI [17] and a number of partitioned global address space (PGAS) languages like UPC [7], Titanium [22], and Co-Array Fortran [19] use the SPMD model by default. Previous work on Titanium also shows that the simplicity of the SPMD model can be used avoid certain classes of deadlocks, statically detect data races, and perform a set of optimizations specific to the parallel setting [14, 15].

While SPMD has proven to be a valuable programming model, its restrictiveness does have drawbacks. Algorithms that divide tasks among threads or that recursively subdivide do not fit well into a model with a fixed number of threads executing the same code. SPMD programming languages also tend to have a relatively flat machine model, with no distinction between threads that are located nearby on a large-scale machine and threads that are further apart. This lack of awareness of the underlying machine hierarchy results in communication costs that are not transparent to the programmer. Similarly, it can be difficult to program for heterogeneous machines using the SPMD model. Finally, SPMD programs have difficulty coping with variable levels of runtime parallelism and cannot easily perform dynamic load balancing.

In this paper, we address some of the above shortcomings by extending the SPMD model with user-defined hierarchical teams, which are subsets of threads that cooperatively execute pieces of code. We introduce a data structure to represent teams, library functions to facilitate team creation, and language constructs that operate on teams. We demonstrate how to ensure textual alignment of collectives, eliminating many forms of deadlock involving teams. Our implementation is in the context of the Titanium programming language, and we evaluate the language additions through case studies on two applications. We demonstrate that hierarchical teams are useful for expressing computations that naturally subdivide, without having to create and destroy logical threads, retaining the advantages of parallel thinking. We also show that hierarchical teams can be used to optimize for the varying communication characteristics of modern, hierarchical parallel machines, achieving significant performance improvements.

2. Background

The single program, multiple data (SPMD) model of parallelism consists of a set of parallel threads that run the same program. Unlike in dynamic task parallelism, the set of threads is fixed throughout the entire program execution. The threads can be executing at different points of the program, though *collective operations* such as barriers can synchronize the processes at a particular point in the program.

As an example of SPMD code, consider the following written in the Titanium language:

```
public static void main(String[] args) {
  System.out.println("Hello from thread " +
                     Ti.thisProc());
  Ti.barrier();
  if (Ti.thisProc() == 0)
    System.out.println("Done.");
}
```

A fixed number of threads, specified by the user on program start, all enter `main()`. They first print out a message with their thread IDs, which can appear to the user in any order since the print statement is not synchronized. Then the threads execute a *barrier*, which prevents them from proceeding until all threads have reached it. Finally, thread 0 prints out another message that appears to the user after all previous messages due to the barrier synchronization.

SPMD languages occupy a middle ground between task and data parallelism; they are more structured and therefore easier to program and more analyzable than task parallel languages, but are more flexible and provide better performance than data parallel languages.

Prior work has shown the benefit of assuming textual alignment of collectives [14]. Collectives are *textually aligned* if all threads execute the same textual sequence of collective operations. For example, the following code violates textual alignment, since different threads take different branches and therefore reach different textual instances of a collective:

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

Discussions with parallel application experts indicate that most applications do not contain unaligned collectives, and most of those that do can be modified to do without them. Our own survey of eight NAS Parallel Benchmarks [4] using MPI demonstrated that all of them only use textually aligned collectives. Prior work has also demonstrated how to enforce textual collective alignment using dynamic checks [16].

2.1 Titanium

Titanium is an explicitly parallel dialect of Java that uses the SPMD model of parallelism and the PGAS memory model. The partitioned global address space (PGAS) model allows any thread to directly access memory on other threads, though with a performance penalty. As an example, consider the following Titanium code:

```
int[] local mydata = { 0, 1, ... };
int[] data0 = broadcast mydata from 0;
for (int i = 0; i < data0.length; i++)
  ... data0[i] ...
```

In this code, each thread creates an integer array in its own memory space. Thread 0 then broadcasts a pointer to its array to the other threads, which can then access elements of thread 0's array, albeit with a possible performance penalty.

As can be seen in the example above, PGAS languages tend to expose some degree of memory hierarchy to the programmer. In Titanium, pointers can be *thread local*, *node local*, or *global*. Thread local pointers can only address data on the same thread[1], node local pointers can only reference data in the same physical address space, and global pointers can point to any object in the program. By default, pointers in Titanium are global, and the `local` qualifier specifies that a pointer is node local. Other PGAS languages such as UPC only have two levels of hierarchy.

While Titanium does have a memory hierarchy, like most other SPMD languages, it does not have a concept of execution hierarchy. Some languages such as UPC are moving towards an execution model based on teams [3], in which the set of program threads can be divided into smaller subsets that cooperatively run pieces of code. The GASNet [6] runtime layer used in Titanium now has experimental support for teams and team collectives. Unlike the teams in our work, teams in both UPC and GASNet are non-hierarchical groupings of threads.

Previous work has shown that PGAS languages in general [23] and Titanium specifically [24] provide an excellent combination of application performance and programmer productivity, justifying our decision to pursue hierarchical extensions to parallel programming in the context of the Titanium language.

2.2 Related Work

Many current languages besides the SPMD languages mentioned above are locality-aware, supporting two levels of machine hierarchy. In the X10 language [20], the memory and space is composed of *places*, and tasks execute at specific places. Remote data can only be accessed by spawning a task at the target place. Chapel [8] has a similar concept of *locales*, and it allows data structures to be distributed across locales. Data parallel operations over such data structures spawn tasks at each locale to locally operate on data, and tasks can also be spawned at particular locales.

Only a handful of existing parallel languages incorporate hierarchical programming constructs beyond two levels of hierarchy.

In the Fortress language [2], memory is divided into an arbitrary hierarchy of *regions*. Data structures can be spread across multiple regions, and tasks can be placed in particular regions by the programmer.

The Sequoia project [10] incorporates machine hierarchy in its language model. A Sequoia program consists of a hierarchy of tasks that get mapped to the computational units in a hierarchical machine. Sequoia has two types of tasks: *inner tasks* that decompose computations into subtasks and *leaf tasks* that perform actual computation. Both the height and width of the resulting task hierarchy can be controlled by the user when starting the program. Communication between tasks is very limited: only parent and child tasks can communicate, through the use of parameters. This restriction on communication as well as the lack of collective operations make the Sequoia model unsuitable for many applications written in SPMD and PGAS languages.

The hierarchical place trees abstraction [21] extends the Sequoia model to allow more general communication between tasks and incorporates X10's ability to spawn tasks at specific locations in a machine. The programming model, however, is still essentially task parallel, with task queues at each location to run tasks. This model both lacks the simple, analyzable structure of SPMD parallelism and the latter's mechanisms for cooperative synchronization and communication.

Hierarchically tiled arrays (HTAs) [5] allow data structures to be hierarchically decomposed to match a target machine's layout. A program can then operate on these data structures in an essentially data parallel manner, with the compiler and runtime mapping execution according to the data layout. Like other data parallel languages, however, the HTA model is quite restrictive, as it is difficult to write applications with irregular task or communication structures.

The HotSLAW library for UPC [18] extends the SPMD model of UPC with dynamic task parallelism. Dynamically created tasks are executed using task queues on a subset of the UPC threads. HotSLAW includes a hierarchical work stealing algorithm using user-defined, hierarchical locality domains for load balancing. Hierarchical load balancing [25] has also been implemented in the context of the Charm++ programming language [12].

The concept of thread teams has been gaining popularity in the SPMD community. MPI has communicators that allow a subset of threads to perform collective operations, and other communication layers and programming languages have recently introduced or are in the process of introducing similar team constructs. However, MPI communicators place no restriction on the underlying thread structure of a team, and a thread can be a part of multiple communicators concurrently, making it easy to deadlock a program through improper use of communicators. Even correct use of multiple communicators can be difficult for programmers to understand and compilers to analyze, as they must reason about the order of communicator calls on each thread. Finally, communicators do not have a hierarchical structure, so they cannot easily reflect the layout of the underlying machine.

3. Language Extensions

Before discussing the Titanium team extensions, we first give an overview of the design philosophy behind the additions. We then

[1] Thread local pointers in Titanium are actually only used by program analysis and are not exposed in the type system.

describe the team representation, followed by the new language constructs that operate on teams.

3.1 Design Goals

In designing the new additions to the Titanium language, we had a few goals in mind for the extensions to satisfy: safety, composability, support for collectives, and performance.

1. **Safety.** Team implementations in other SMPD languages and frameworks do not generally impose any restrictions on their use. This can lead to circular dependencies in team operations, resulting in deadlock. For example, a set of threads may attempt to perform a collective operation on one team, while other threads that they depend on attempt to perform a collective operation on a different team that overlaps with the first set. The Titanium team extensions should prevent such circular dependencies. It also should ensure that team collectives are textually aligned on all threads in the relevant team, as it does for existing global collectives.
2. **Composability.** Existing code running in the context of a particular team should behave as if the entire world consisted of just the threads in that team, with thread ranks as specified by the team. This is to facilitate composition of different tasks, so that a subset of threads can be assigned to each of them. At the same time, the team library should provide new programs with the ability to interact with threads outside of an assigned team.
3. **Support for Collectives.** One of the key features of the SPMD programming model is the ability of threads to communicate and synchronize through collective operations, such as reductions and barriers. Without support for collective operations over teams, users would have to hand-write their own implementations, requiring extensive development time and resulting in suboptimal performance, as in the conjugate gradient application described in §5.2.
4. **Performance.** Team operations should not adversely affect application performance. This requires that team usage operations, which may be invoked many times throughout an application run, be as lightweight as possible, even at the expense of team creation operations that are called much less frequently.

3.2 Team Representation

In order to represent a team hierarchy, we introduced a new `Team` object, as shown in Figure 1. A `Team` represents a portion of a team hierarchy, containing references to its parent and child teams as well as its member threads. Like MPI or GASNet groups, `Team` objects specify team structure without forcing a program to actually execute using that structure; this is useful when a program uses multiple different team structures or repeatedly uses the same structure, as in §5.2, and also allows team data structures to be manipulated as first-class objects.

Knowledge of the physical layout of threads in a program allows a programmer to minimize communication costs, so a new method `Ti.defaultTeam()` returns a special team that corresponds to the layout of threads at runtime. Currently, it merely groups together threads that share memory, though future use of the *hwloc* library [1] will provide a more representative layout. The invocation `Ti.currentTeam()` returns the current team in which the calling thread is participating.

Figure 2 shows the team hierarchy created by the following code, when there are a total of twelve threads:

```
Team t = new Team();
t.splitTeam(3);
int[][] ids = new int[][] {{0, 2, 1}, {3}};
for (int i = 0; i < t.numChildren(); i++)
  t.child(i).splitTeamRelative(ids);
```

```
public class Team {
  // Create team with all threads in currently
  // executing team.
  public Team();
  // Returns the ith child of this team.
  public Team child(int i);
  // Number of child teams.
  public int numChildren();
  // Rank of this team in its parent.
  public int teamRank();
  // Number of threads in this team.
  public int size();
  // The child team containing the calling thread.
  public Team myChildTeam();
  // Split team into n equally-sized subteams,
  // with threads retaining relative ranks.
  public void splitTeam(int n);
  // Split team into n child teams, with threads
  // assigned to subteams in block cyclic order.
  public void splitTeamBlockCyclic(int n, int sz);
  // Split team into the given subteams, with
  // ranks specified relative to this team.
  public void splitTeamRelative(int[][] teams);
  // Collective split operation. Assigns threads
  // to subteams according to color and the given
  // rank relative to other threads.
  public single void splitTeamAll(int color,
                                  int relrank);
  // Collective split operation. Divides threads
  // into subteams of threads that share memory,
  // with the given relative rank.
  public single void splitTeamSharedMem(int rel);
  // Collective operation. Constructs a new team
  // in which each subteam consists of a single
  // thread from each subteam of this team.
  public single Team single makeTransposeTeam();
  // Initialize runtime structures required by
  // this team and run consistency checks.
  public single void initialize(boolean check);
}
```

Figure 1. Relevant functions from the Titanium `Team` class.

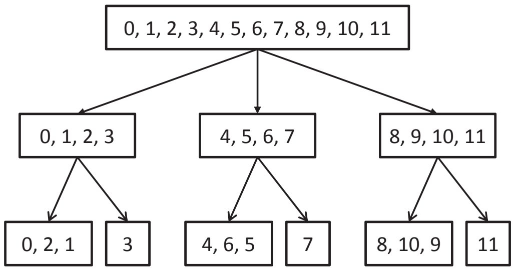

Figure 2. An example of a team hierarchy.

Each box in the diagram corresponds to a node in the team tree, and the entries in each box refer to member threads by their global ranks.

The code above first creates a team consisting of all the threads and then calls the `splitTeam` function to divide it into three equally-sized subteams of four threads each. It then divides each of those subteams into two uneven, smaller teams. Note that since the IDs used in `splitTeamRelative` are relative to the parent team, the same code can be used to divide each of the three teams at the second level, which would not be the case if the IDs were global.

The `Team` class provides a few other ways of generating subteams, as shown in Figure 1. In addition, it includes numerous functions to query team properties, a sample of which are also shown in Figure 1.

Before using teams with the constructs introduced below, the programmer must call `initialize`, which is a collective operation that performs the runtime setup needed by a team and checks team consistency across threads. In our current implementation, this initialization is separate from team creation, allowing a user to construct a team on a single thread and then broadcast it to the others. Those threads then must create local copies since `Team` objects contain thread-specific state. So far, we have not found this to be a useful feature, and we may remove this functionality in order to combine team creation and initialization.

3.3 New Language Constructs

In designing new language constructs that make use of teams, we identified two common usage patterns for grouping threads: sets of threads that perform different tasks and sets of threads that perform the same operation on different pieces of data. We introduced a new construct for each of these two patterns.

Task Decomposition. In task parallel programming, it is common for different components of an algorithm to be assigned to different threads. For example, a climate simulation may assign a subset of all the threads to model the atmosphere, another subset to model the oceans, and so on. Each of these components can in turn be decomposed into separate parts, such as one piece that performs a Fourier transform and another that executes a stencil. Such a decomposition does not directly depend on the layout of the underlying machine, though threads can be assigned based on machine hierarchy.

Task decomposition can be expressed through the following *partition* statement that divides the current team into subteams:

`partition` (T) { B_0 B_1 ... B_{n-1} }

A `Team` object (corresponding to the current team at the top level) is required as an argument. The first child team executes block B_0, the second block B_1, and so on. It is an error if there are fewer child teams than partition branches, or if the given team arguments on each thread in the current team do not have the same description of child teams. If the provided team has more than n subteams, the remaining subteams do not participate in the partition construct. Once the partition is complete, threads rejoin the previous team.

As a concrete example, consider a climate application that uses the team structure in Figure 2 to separately model the ocean, the land, and the atmosphere. The following code would be used to divide the program:

```
partition(t) {
  { model_ocean(); }
  { model_land(); }
  { model_atmosphere(); }
}
```

Threads 0 to 3 would then execute `model_ocean()`, threads 4 to 7 would run `model_land()`, and threads 8 through 11 would model the atmosphere.

Since partition is a syntactic construct, task structure can be inferred directly from program structure. This simplifies program analysis and improves understandability of the code.

Data Decomposition. In addition to a hierarchy of distinct tasks, a programmer may wish to divide threads into teams according to algorithmic or locality considerations, but where each team executes the same code on different sets of data. Such a data decomposition can be either machine-dependent or required by an algorithm, and both the height and width of the hierarchy may differ according to the machine or algorithm.

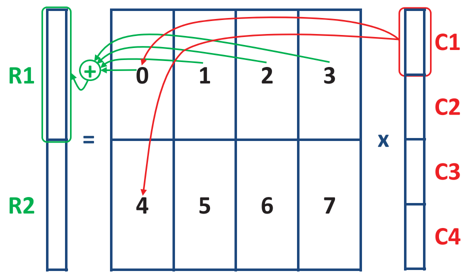

Figure 3. Blocked matrix-vector multiplication.

Consider the matrix-vector multiplication depicted in Figure 3, where the matrix is divided in both dimensions. In order to compute the output vector, threads 0 to 3 must cooperate in a reduction to compute the first half of the vector, while threads 4 to 7 must cooperate to compute the second half. Both sets of threads perform the same operation but on different pieces of data.

A new *teamsplit* statement with the following syntax allows such a data-driven decomposition to be created:

`teamsplit` (T) B

The parameter T must be a `Team` object (corresponding to the current team at the top level), and as with partition, all threads must agree on the set of subteams. The construct causes each thread to execute block B as a member of its new subteam specified by the team argument.

Common Features. Both the partition and teamsplit constructs are lexically scoped, changing the team in which a thread is executing within that scope. This implies that at any point in time, a thread is executing in the context of exactly one team (which may be a subteam of another team). Given a particular team hierarchy, entering a teamsplit or partition statement moves one level down in the hierarchy, and exiting a statement moves one level up. Statements can be nested to make use of multi-level hierarchies, and recursion can be used to operate on hierarchies that do not have a pre-determined depth. Consider the following code, for example:

```
public void descendAndWork(Team t) {
  if (t.numChildren() != 0)
    teamsplit(t) {
      descendAndWork(t.myChildTeam());
    }
  else
    work();
}
```

This code descends to the bottom of an arbitrary team hierarchy before performing work. A concrete example that uses this paradigm is provided in §5.3.

In order to meet the composability goal of §3.1, the thread IDs returned by `Ti.thisProc()` are now relative to the team in which a thread is executing, and the number of threads returned by `Ti.numProcs()` is equal to the size of the current team. Thus, a thread ID is always between 0 and `Ti.currentTeam().size()`-1, inclusive. A new function `Ti.globalNumProcs()` returns the number of threads in the entire program, and `Ti.globalThisProc()` returns a thread's global rank.

Collective communication and synchronization now operate over the current team. The combination of the requirement that all threads must agree on the set of subteams when entering a partition or teamsplit construct, lexical scoping of the constructs, and textual collective alignment ensure that no circular dependencies exist between different collective operations. In §4, we describe how the first and last properties are enforced.

Discussion. It may be apparent that the partition statement can be implemented in terms of teamsplit, such as the following:

```
teamsplit(t) {
  switch(t.myChildTeam().teamRank()) {
  case 0:
    model_ocean();
    break;
  case 1:
    model_land();
    break;
  case 2:
    model_atmosphere();
  }
}
```

While this is true, we decided that an explicit construct for task decomposition is cleaner and more readable than the combination of teamsplit and branching. The two constructs also differ with respect to the superset operations described below.

Superset Operations. By design, the partition and teamsplit constructs require a user to exit or enter a construct to move up or down a team hierarchy or to use multiple team hierarchies. We suspect that it may be useful, however, to be able to temporarily move up one or more levels in a team hierarchy without exiting a partition or teamsplit, though we have yet to find concrete examples where this is the case. Nevertheless, our implementation contains a *superset* statement that ascends the team hierarchy within a specified lexical scope:

superset(i) B

This results in execution of block B in the context of the team that is i levels up from the enclosing team, i.e. that team's ith ancestor. Since teams in a partition execute different code, the i enclosing team statements must all be teamsplits in order to conform to textual collective alignment. It is an error if all threads in the ith ancestral team do not execute the superset, if they differ on the value of i, or if no ith ancestor exists. Superset statements may be nested, but they may not contain any teamsplit or partition statements.

We anticipate that the most likely use of a superset operation will be to perform a collective, such as a barrier, at a higher level in the team hierarchy. As such, we have implemented versions of many collective operations that operate at higher levels without requiring an explicit superset construct. For example, the call `Ti.barrier(1)` executes a barrier on the parent team. Of course, such operations must meet the same requirements as the superset construct itself.

Implementation. We have implemented hierarchical team constructs on top of GASNet teams, which are flat groupings of threads. Each node in a team hierarchy is associated with a separate GASNet team, which are created when teams are initialized. When changing team contexts by entering or exiting a teamsplit, partition, or superset operation, the Titanium runtime sets the current GASNet team on each thread accordingly and updates a handful of variables to reflect the properties of the new team. As a result, there is very low overhead to switching team contexts, satisfying the performance goal of §3.1. In order to execute a collective, the Titanium runtime passes the current GASNet team to the GASNet collectives library, resulting in the desired behavior that collectives only execute over the current team.

4. Alignment of Collectives

Like other SPMD languages, Titanium provides primitive *collective operations* to allow program threads to synchronize and communicate together. These operations include barriers, broadcasts, exchanges, and reductions, as well as the the partition, teamsplit, and superset statements introduced in §3.

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.

4.1 Alignment Rules

Titanium requires that collectives be textually aligned on all threads that participate in a collective. The basic conditions that guarantee this alignment consist of the following global rules:

1. If any branch of a conditional has effects on team t, then all threads in t must take the same branch.
2. If the body/test of a loop has effects on team t, then all threads in t must execute the same number of iterations.
3. If a method call has effects on team t, then the dynamic dispatch target of the call must be the same on all threads in t.
4. The source thread in a broadcast and target thread in a reduction on team t must evaluate to the same value on each thread in t.
5. `Team` objects passed to a partition or teamsplit on team t must have consistent immediate subteams across all threads in t.
6. The level argument passed to a superset statement or collective that targets team t must evaluate to the same value on all threads in t.

In addition, the following local rules must be satisfied:

7. `Team` objects passed to a partition or teamsplit must match the current team at the top level.
8. A superset operation with a level argument i must be enclosed by i teamsplit statements, with no intervening partition statements.

Different definitions of *has effects on team* t result in different enforcement schemes [16]. For example, *weak alignment* results if a statement or expression has effects on team t if it or a substatement or subexpression actually executes a primitive collective operation at runtime. The enforcement extensions we discuss below work for both strict and weak alignment.

4.2 Alignment Enforcement

The first four conditions above are enforced on the global team by the **single** type system in the current version of Titanium or by a system of dynamic checks in an experimental version [16]. The static type system cannot be easily extended to handle all teams [13], so we extended the dynamic version to work on teams.

The basic idea behind the dynamic enforcement system is that each thread tracks all control flow decisions that potentially affect alignment of a collective. Prior to executing a collective, the threads cooperate to perform a global check to ensure that they are all aligned. Since this check runs before each collective, misaligned collectives are never executed, as they are detected in the preceding check. A failed check results in the program terminating with an error message describing the sequence of control flow decisions that resulted in the misalignment. An implicit barrier with a corresponding alignment check occurs at program end, catching any unmatched collectives.

The first four alignment rules are enforced by inserting an entry in a per-thread tracking list recording the decision made when executing an affected program expression or statement. Various optimizations such as hashing can minimize the overhead of dynamic checking [16]. Since partition and teamsplit statements are collectives themselves, rules 5 and 7 can be checked directly when entering such a statement. We come back to rules 6 and 8 later.

In order to extend dynamic enforcement to work on all teams, we now keep separate track of control flow decisions that may affect alignment of collectives on different teams. To simplify the discussion, we ignore the existence of superset operations until later. Then it is sufficient for both strict and weak alignment to record control flow decisions in the tracking list for the current team. Upon encountering a collective operation, the tracking list is checked for consistency among only the threads in the current team. A final check must be made at the end of a partition or teamsplit to ensure that no unmatched collectives exist within such a statement.

As a concrete example, consider the following code:

```
1  if (a) Ti.barrier();
2  teamsplit(u) {
3    if (b) Ti.barrier();
4  }
5  if (c) Ti.barrier();
```

Let t be the current team outside the teamsplit. When the conditional on line 1 is executed, each thread in team t records the branch taken in the tracking list for t. Those threads that take the *if* branch await a check before performing the barrier. If other threads do not take this branch, they perform a check before the teamsplit, resulting in the error being detected. Upon encountering the teamsplit and executing a successful check, the threads in t ensure that team u is equivalent to t at the top level and that the immediate subteams of u are the same on all threads. If this is the case, then each thread's corresponding subteam in u becomes the new current team on that thread, and control flow decisions within the teamsplit are now recorded in the tracking list for `u.myChildTeam()`. Thus, it is perfectly valid for one subteam of u to execute the barrier on line 3 while other subteams skip it. Let v refer to the first subteam of u. Then all threads in v record the branch taken, and if some threads in v take the *if* branch, they will await a check before performing the barrier. If other threads do not take the branch, then they will perform a check at the end of the teamsplit, resulting in detection of the error. Finally, upon leaving the teamsplit, t once again becomes the currently executing team, so that alignment of the barrier on line 5 will be checked with respect to team t.

Note that it is always necessary to ensure that alignment is consistent in the current team before entering a new team context. The checks prior to teamsplit and partition and at their end ensure that this is the case.

Superset Operations. Superset operations complicate alignment checking since they take in a level parameter that may not be known at compile-team. Consider the following code:

```
if (d) {
  int n = ...; // not a compile-time constant
  Ti.barrier(n);
  Ti.barrier(n+1);
}
```

Prior to entering the branch, a thread may not be able to determine at what levels the barriers execute and thus which teams' tracking lists need to be updated. In order to handle this, an orphan tracking list must be maintained that keeps track of control flow decisions that may affect alignment of a superset operation. Then when such an operation is encountered, the orphan list's contents are copied into that of the teams affected by the operation.

A superset operation is defined to have team effects on all teams between the current team and the target team[2]. Thus, in checking a superset operation, alignment must first be checked at the current team level and all intermediate levels up to the target level, in order, to ensure that alignment is satisfied at every level. Then rules 6 and 8 in §4.1 are checked before performing the superset operation.

[2] In strict alignment, a superset operation that is not executed at runtime is defined to have effects only on the current team, since the target team is unknown.

4.3 Implementation

We have implemented team extensions to dynamic enforcement in the context of weak alignment. Dynamic enforcement results in runtime overhead; in particular, simple collective operations such as broadcasts can take twice as long on larger numbers of threads [16]. As with other runtime checks such as array bounds checking, Titanium allows a user to turn them off in order to eliminate their overhead. Most users enable checks when debugging but disable them in production runs, as we have done in §5.

5. Application Case Studies

In order to guide the design of the language constructs described in §3 and evaluate their effectiveness, we examined two application benchmarks to determine how they can benefit from the new hierarchical team constructs. In this section, we present case studies of the two applications, conjugate gradient and parallel sort.

5.1 Test Platforms

We tested application performance on two machines, a Cray XT4 and a Cray XE6. The Cray XT4 is a cluster of quad-core AMD Budapest 2.3 GHz processors, with one quad-core processor per node. The Cray XE6 consists of two twelve-core AMD MagnyCours 2.1 GHz processors per node, each of which consists of two six-core dies, also called non-uniform memory access (NUMA) nodes since each die has fast access to its own memory banks but slower access to the other banks. On both machines, we used the MPI conduit of GASNet for communication.

5.2 Conjugate Gradient

The conjugate gradient (CG) application is one of the NAS parallel benchmarks [4]. It iteratively determines the minimum eigenvalue of a sparse, symmetric, positive-definite matrix. The matrix is divided in both dimensions, and each thread receives a contiguous block of the matrix, with threads placed in row major order. The application performs numerous sparse matrix-vector multiplications. Consider a blocked matrix-vector multiplication, as illustrated previously in Figure 3. Each element in the source vector must be distributed to the threads that own a portion of the corresponding matrix column. Each element in the destination vector is computed using a reduction across the threads that own a portion of the corresponding matrix row.

Original Implementation. Prior to our language extensions, Titanium only supported collectives over all threads in a program. Thus, the original Titanium implementation of CG [9] required hand-written reductions over subsets of threads. These reductions required extensive development effort to implement, test, and optimize.

The original implementation performs an all-to-all reduction on each row, so that in the example of Figure 3, threads 0 to 3 receive the first half of the result vector and threads 4 to 7 receive the second half. Since the algorithm is iterative, the result vector becomes the input vector in the next iteration, so that threads 0 and 4 require the first quarter of the vector, threads 1 and 5 the second quarter, and so on. In order to accomplish this, threads 2 and 4 swap their results, as do threads 3 and 5. The remaining threads already have their required data.

Row Teams. The first step in modifying CG to use teams was to replace the hand-written all-to-all reductions with built-in reductions on teams. The code already computes the row and column number of each thread, which we use to divide the threads into row teams with a call to **splitTeamAll**. Then in each matrix-vector multiplication, a single library call is all that is necessary to perform a reduction across each row team, as shown in the below code.

```
Team rowTeam = new Team();
public void initialize() {
  rowTeam.splitTeamAll(rowPos, colPos);
  rowTeam.initialize(false);
  ...
}
public void multiply(Vector in, Vector out) {
  ...
  teamsplit(rowTeam) {
    Reduce.add(tmp, myResults);
  }
  ...
}
```

Column Teams. The above implementations perform unnecessary communication, implicitly broadcasting the results of a row reduction to the entire row before transposing the result vector across columns. In order to further optimize the code, we replaced the all-to-all reductions with all-to-one reductions, so that in the example of Figure 3, only thread 0 receives the first half of the result vector and thread 6 receives the second half of the result vector. Thread 1 then copies the portion of the result vector needed by the second column from thread 0, and thread 7 similarly copies from thread 6. Now that one thread in each column has the required data, it broadcasts the result to the remaining threads in that column.

As before, a team for each row is required to perform the reductions. In addition, a team for each column is required to perform the broadcasts. Lastly, we construct additional teams to synchronize the source and destination threads of the copies between the reductions and broadcasts. The below code demonstrates these operations.

```
  teamsplit(rowTeam) {
    Reduce.add(tmp, myResults, rpivot);
  }
  teamsplit(copyTeam) {
    if (copySync)
      Ti.barrier();
  }
  if (reduceCopy)
    myOut.copy(allResults[reduceSource]);
  teamsplit(columnTeam) {
    myOut.vbroadcast(cpivot);
  }
```

Evaluation. The CG application demonstrates the importance of teams for collective operations among subsets of threads. It also illustrates the need for multiple team hierarchies and for separating team creation from usage, as the cost for creating teams is amortized over all iterations of the algorithm.

Figures 4 and 5 compare the performance of the three versions of CG on a Cray XT4 and a Cray XE6. We show strong scaling (fixed problem size) results using two problem sizes, Class B for one to 128 threads and Class D for 128 to 1024 threads. (Note that both axes in the figures use logarithmic scale, so ideal scaling would appear as a line on the graphs.) As expected, the replacement of hand-written reductions with optimized GASNet reductions in the row teams version improves performance over the original version. The communication optimizations resulting from the addition of column teams further improves performance, achieving speedups over the original code of 2.1x for Class B at 128 threads and 1.6x for Class D at 1024 threads on the XT4. The XE6 shows similar speedups of 1.6x and 1.5x for the same problem sizes and thread counts.

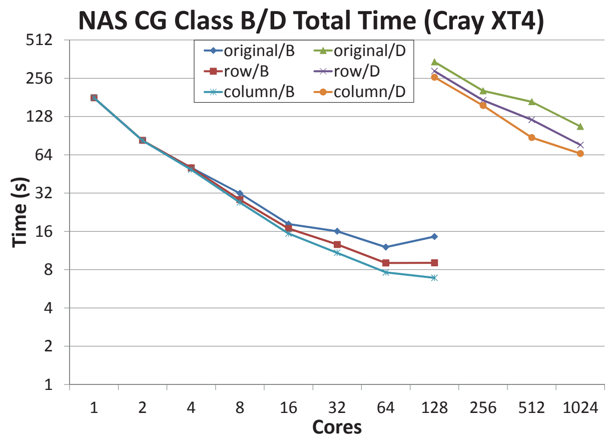

Figure 4. Strong scaling performance of conjugate gradient on Cray XT4.

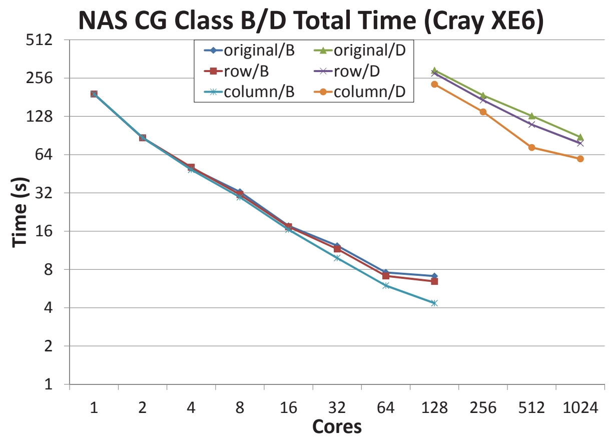

Figure 5. Strong scaling performance of conjugate gradient on Cray XE6.

As for parallel scaling, the column team version achieves a speedup of 26x for Class B on 128 threads over the sequential version on the XT4 and 44x on the XE6. Note that it should be no surprise that the implementation ceases to scale for Class B, since communication time dominates computation time for this problem size at higher numbers of threads. For Class D, we achieve a speedup of about 4x on 1024 threads over 128 threads on both machines.

Shared Memory Optimizations. In addition to rewriting the CG code to use team collectives, we wrote an experimental version that explores various shared memory optimizations. In particular, if threads that share memory are placed in the same column, these threads can share their portion of the input vector. As a result, the broadcasts described in §5.2 can be restricted to one thread on each node, further reducing the amount of communication.

In implementing this optimization, we first had to divide the matrix among threads in column major order. Unfortunately, the assumption of row major order is pervasive in the code, so directly making this change proved difficult. Instead, we constructed a new global team that merely rearranged thread IDs to produce the desired ordering. We then called the unmodified CG code in the context of this team, as follows:

```
public static void main(String[] args) {
  ... // Compute number of rows and columns.
  int id = Ti.thisProc();
  id = id / numRows + numCols * (id % numRows);
  Team flipTeam = new Team();
  flipTeam.splitTeamAll(0, id);
  flipTeam.initialize(false);
  teamsplit(flipTeam) {
    oldMain(args);
  }
}
public static void oldMain(String[] args) {
  ...
}
```

We then had to divide each column team into subteams of shared memory threads to allow for synchronization on their shared piece of the input vector. We also had to construct teams with one thread from each node in a column in order to perform the broadcasts. The below code constructs both sets of teams.

```
  teamsplit(columnTeam) {
    colSMPTeam = new Team();
    colSMPTeam.splitTeamSharedMem(id);
    colSliceTeam = colSMPTeam.makeTransposeTeam();
    colSMPTeam.initialize(false);
    colSliceTeam.initialize(false);
  }
```

Unfortunately, we determined that in most cases, the added synchronization overhead was greater than the time saved by reduced communication in the current version of the code. Despite the disappointing performance results, this exercise demonstrates the benefits of the new team constructs in exploring optimizations, and we plan to investigate whether or not further optimizations can make this version of the code more efficient.

5.3 Parallel Sort

The second application we examined was a sorting library that sorts 32-bit integers in parallel. We postulated that the most efficient implementation would be a hierarchical distributed sort that uses a communication-optimized algorithm between threads that do not share memory but otherwise takes advantage of shared memory. We started with two existing implementations, the sequential quicksort from the `java.util.Arrays` class in the Java 1.4 library and a distributed sample sort written in Titanium by Kar Ming Tang.

Overview of Sample Sort. In the sample sort algorithm [11], data is initially randomly and evenly distributed among all processors. At the end of the algorithm, the elements satisfy two properties: (1) all elements on an individual processor are in sorted order and (2) all elements on processor i are less than any element on processor $i+1$. The algorithm accomplishes this by first randomly sampling the elements on each of the n processors, sending them to processor 0. Processor 0 sorts the samples, determining $n-1$ pivots that divide the elements into n approximately equal sets. Each processor then uses these pivots to divide its elements into n buckets, which are then exchanged among all processors to satisfy property (2) above. Then each processor sequentially sorts its resulting elements to satisfy property (1), and the algorithm terminates.

The bucket exchange operation requires $n(n-1)$ messages, since each processor must send $n-1$ buckets to a different processor. On a cluster of multiprocessors, however, we speculated that it would be more efficient to aggregate communication by using only a single bucket for each node, so that $m(m-1)$ messages would be required for m nodes. Each node's data could then be sorted in parallel by the processors on that node.

Shared Memory Sort. The original implementation of sample sort treats all threads as if they do not share memory. In order to remedy this, we first wrote a new sort implementation that assumes that all threads share memory, relying heavily on the new team constructs in writing the code.

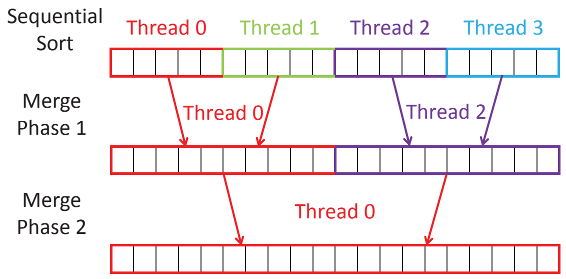

Figure 6. Shared memory sorting algorithm on four threads.

The shared memory algorithm starts with a single, contiguous array of integers. This array is divided equally among all threads, each of which calls the sequential quicksort to sort its elements in-place. The separately sorted subsets are then merged in parallel in multiple phases, with the number of participating threads halved in each stage. Figure 6 illustrates this process on four threads.

The recursive nature of the sorting can be easily represented with a team hierarchy consisting of a binary tree, in which each node contains half the threads as its parent. The following code constructs such a hierarchy, using the `splitTeam()` library function to divide a team in half.

```
static void divideTeam(Team t) {
  if (t.size() > 1) {
    t.splitTeam(2);
    divideTeam(t.child(0));
    divideTeam(t.child(1));
  }
}
```

Then each thread walks down to the bottom of the team hierarchy, sequentially sorts its elements, and then walks back up the hierarchy to perform the merges. In each internal team node, a single thread merges the results of its two child nodes before execution proceeds to the next level in the hierarchy. The following code performs the entire algorithm, and Figure 7 illustrates the process on six threads.

```
static single void sortAndMerge(Team t) {
  if (Ti.numProcs() == 1) {
    allRes[myProc] = SeqSort.sort(myData);
  } else {
    teamsplit(t) {
      sortAndMerge(Ti.currentTeam());
    }
    Ti.barrier(); // ensure prior work complete
    if (Ti.thisProc() == 0) {
      int otherProc = myProc + t.child(0).size();
      int[1d] myRes = allRes[myProc];
      int[1d] otherRes = allRes[otherProc];
      int[1d] newRes = target(t.depth(), myRes,
                              otherRes);
      allRes[myProc] = merge(myRes, otherRes,
                             newRes);
    }
  }
}
```

As illustrated in the above code, the shared memory sorting algorithm is very simple to implement using the new team constructs. The entire implementation is only about 90 lines of code (not including test code and the sequential quicksort) and took just two hours to write and test.

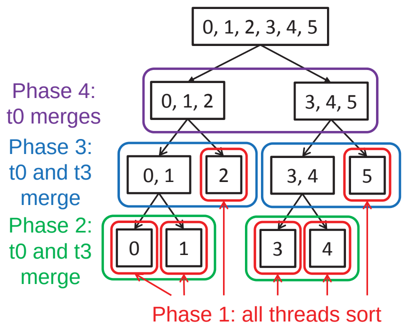

Figure 7. Shared memory team hierarchy and execution on six threads.

Distributed Sort. As an initial hierarchical, distributed sort implementation, we started with an unoptimized sample sort implementation written by Kar Ming Tang in 1999. Our initial implementation assigns a single thread from each node to participate in the sample sort, so that the number of messages is minimized as described previously. Then each node performs the shared memory sort described above. The entire code to accomplish this is as follows:

```
Team team = Ti.defaultTeam();
team.initialize(false);
Team oTeam = team.makeTransposeTeam();
oTeam.initialize(false);
partition(oTeam) {
  { sampleSort(); }
}
teamsplit(team) {
  keys = SMPSort.parallelSort(keys);
}
```

Again, the new team constructs make this algorithm trivial to implement, requiring only 10 lines of code and 5 minutes of development time. The code calls `Ti.defaultTeam()` to obtain a team in which threads are divided according to which threads share memory. It then uses the `makeTransposeTeam()` library call to construct a transpose team in which each subteam contains one thread from each node. The `partition` construct is then used to perform the sample sort on one of those subteams, after which the node teams execute the shared memory sort.

This example illustrates the value of the composability features of the team extensions. As far as the code in `sampleSort()` is concerned, its entire world consists of just a single thread from each node. The only change required was to remove the call to sequential sort after the sampling and distribution. Similarly, as far as the shared memory sort is concerned, its entire world consists of the threads on a single node. No changes were required, and the team hierarchy constructed in the shared memory sort composes cleanly with the hierarchy used here.

Figure 8 illustrates the weak scaling (problem size proportional to number of threads) performance of this initial implementation on a Cray XT4 compared to a pure sample sort. Sorting takes longer in the mixed, hierarchical version, since it is done in parallel, with threads idling in the merge phases. However, the distribution portion of the algorithm takes approximately the same time in both versions at 8 nodes. We speculated that the distribution time in the mixed version would take less time than the pure version at larger numbers of nodes.

The initial implementation does not scale beyond 8 nodes in either the pure sample sort or the mixed, hierarchical version, so we completely reimplemented the sample sort. We omit the details here, but the new version uses all threads to help with sampling

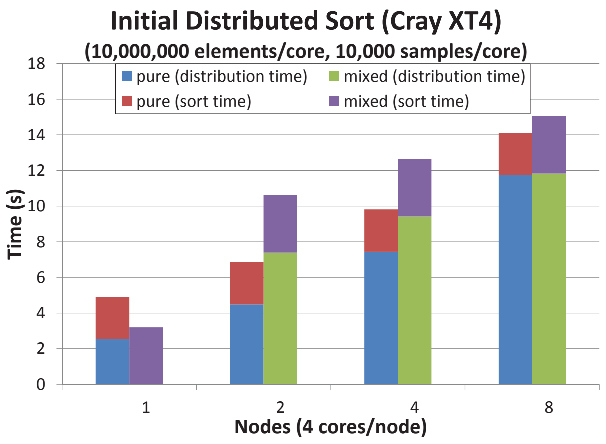

Figure 8. Weak scaling performance of initial distributed sort on Cray XT4.

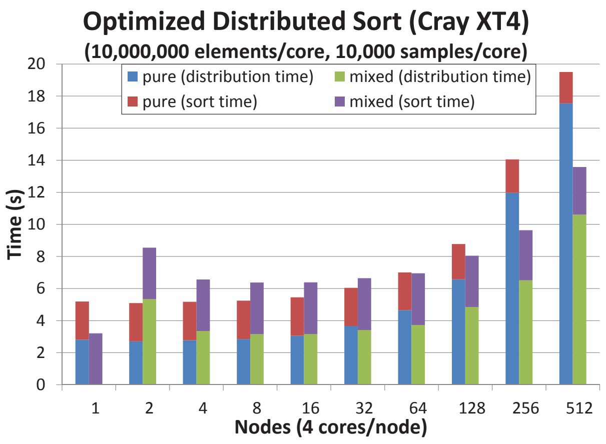

Figure 9. Weak scaling performance of optimized distributed sort on Cray XT4.

and distribution rather than a single thread per node in the mixed version of the code. Communication, however, is still aggregated at the node level to minimize the number of messages required. Figures 9 and 10 compare the performance of pure sample sort and mixed, hierarchical sample and shared memory sort, with both versions using the new sample and distribution code. On the Cray XE6, we used a single Unix process per NUMA node, since its non-uniform memory access makes it inefficient to rely on shared memory between NUMA nodes. On both machines, the gap in distribution time between the pure and mixed versions grows as the number of threads increases, resulting in a speedup of 1.4x for the mixed version over the pure version on both 512 nodes (2048 cores) of the XT4 and 512 NUMA nodes (3072 cores) of the XE6.

As for overall scaling of the algorithm, since the number of elements per node is constant, we expect the sorting phase to remain constant over all numbers of threads, as is the case for both implementations. The distribution phase is dominated by the bucket exchange operation described in §5.3. While the amount of communication per thread remains constant, the total amount of communication increases linearly and the number of messages increases quadratically, accounting for the increase in distribution time at higher numbers of threads.

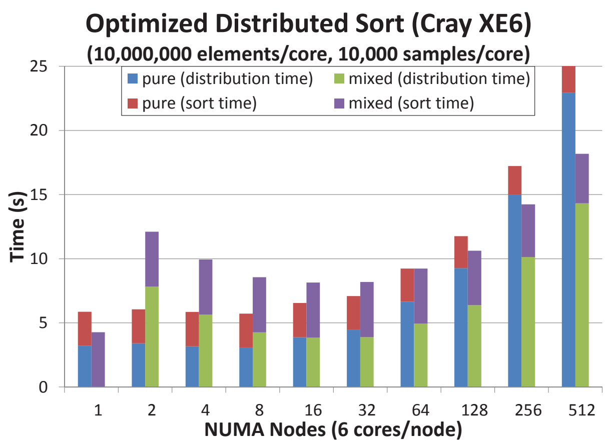

Figure 10. Weak scaling performance of optimized distributed sort on Cray XE6.

6. Conclusion

In this paper, we presented a set of language extensions for programming in a hierarchical manner in the SPMD model of parallelism. We introduced a new team data structure to hierarchically group threads as well as new language constructs to work on teams. We demonstrated how to enforce textual alignment of collectives on teams, eliminating a class of deadlocks. Our experiences with the conjugate gradient and sorting applications demonstrate that the language additions allow simple expression of algorithms that subdivide computation. They also allow programmers to optimize for the communication characteristics of hierarchical machines, taking advantage of shared memory and achieving significant performance gains without resorting to using two different programming models as is typical with MPI.

References

[1] Portable hardware locality (hwloc). URL http://www.open-mpi.org/projects/hwloc/.

[2] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification, Version 0.866*. Sun Microsystem Inc., Feb. 2006.

[3] G. Almási, P. Hargrove, I. G. Tănase, and Y. Zheng. UPC collectives library 2.0. In *Fifth Conference on Partitioned Global Address Space Programming Models*, Galveston Island, Texas, October 2011.

[4] 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. URL citeseer.nj.nec.com/article/bailey94nas.html.

[5] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–57, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: http://doi.acm.org/10.1145/1122971.1122981.

[6] D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, November 2002.

[7] 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.

[8] Cray Inc. *Chapel Specification 0.4*, Feb. 2005.

[9] 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.

[10] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. pages 4–4, Nov. 2006. doi: 10.1109/SC.2006.55.

[11] J. Huang and Y. Chow. Parallel sorting and data partitioning by sampling. In *7th International Computer Software and Applications Conference*, 1983.

[12] L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993.

[13] A. Kamil. Problems with the titanium type system for alignment of collectives, February 2006. http://www.cs.berkeley.edu/~kamil/titanium/doc/single.pdf.

[14] 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.

[15] A. Kamil and K. Yelick. Hierarchical pointer analysis for distributed programs. In *The 14th International Static Analysis Symposium (SAS 2007, Kongens Lyngby*, August 2007.

[16] A. Kamil and K. Yelick. Enforcing textual alignment of collectives using dynamic checks. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, October 2009.

[17] Message Passing Interface Forum. MPI: A message-passing interface standard, version 1.1, 1995. http://www.mpi-forum.org/docs/.

[18] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models*, Galveston Island, Texas, October 2011.

[19] R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[20] V. Saraswat. *Report on the Experimental Language X10, Version 0.41*. IBM Research, Feb. 2006.

[21] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, October 2009.

[22] 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.

[23] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *Parallel Symbolic Computation 2007*, July 2007.

[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. *The International Journal of High Performance Computing Applications*, 21(2), Summer 2007.

[25] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale. Hierarchical load balancing for Charm++ applications on large supercomputers. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, 2010.