Hierarchical Computation in the SPMD Programming Model

Amir Kamil Katherine Yelick

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

Abstract. Large-scale parallel machines are programmed mainly with the single program, multiple data (SPMD) model of parallelism. While this model has advantages of scalability and simplicity, it does not fit well with divide-and-conquer parallelism or hierarchical machines that mix shared and distributed memory. In this paper, we define the recursive single program, multiple data model (RSPMD) that extends SPMD with a hierarchical team mechanism to support hierarchical algorithms and machines. We implement this model in the Titanium language and describe how to eliminate a class of deadlocks by ensuring alignment of collective operations. We present application case studies evaluating the RSPMD model, showing that it enables divide-and-conquer algorithms such as sorting to be elegantly expressed and that team collective operations increase performance of conjugate gradient by up to a factor of two. The model also facilitates optimizations for hierarchical machines, improving scalability of particle in cell by 8x and performance of sorting and a stencil code by up to 40% and 14%, respectively.

1 Introduction

The single program, multiple data (SPMD) model of parallelism, in which a program is launched with a fixed number of threads that execute throughout the program, is the dominant programming model for large-scale distributed-memory machines. The model encourages "parallel thinking" throughout the program execution, exposing the actual degree of available parallelism, naturally leads to good locality, and can be implemented by simple, low-overhead runtime systems. Both message-passing models like MPI [19] and a number of partitioned global address space (PGAS) languages like UPC [6], Titanium [23], and Co-Array Fortran [20] use the SPMD model by default. Previous work on Titanium also shows that the simplicity of the SPMD model can be used to avoid certain classes of deadlocks, statically detect data races, and perform a set of optimizations specific to the parallel setting [15,16].

While SPMD has proven to be a valuable programming model, the restrictiveness of the flat SPMD model 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 makes it difficult to reason about the communication costs between threads. While some SPMD languages do address these issues with teams

or virtual topologies, they do not do so in a structured manner that provides flexibility and performance but prevents deadlocks.

In this paper, we address the shortcomings above by defining the recursive single program, multiple data (RSPMD) model. This model extends SPMD with user-defined hierarchical teams, which are subsets of threads that cooperatively execute pieces of code. We introduce new language and library features for hierarchical teams and describe 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 on four applications. We demonstrate that hierarchical teams enable the expression of divide-and-conquer algorithms with a fixed number of threads, that team collectives provide better performance than hand-written communication code, and that hierarchical teams allow optimizations for the communication characteristics of modern, hierarchical parallel machines.

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 threads at a particular point in the program.

As an example of SPMD code, consider the following written in the Titanium language[1]:

```
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, or *ranks*, 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.

Prior work has shown the benefit of assuming textual alignment of collectives [15]. Collectives are *textually aligned* if all threads execute the same textual sequence of collective operations, and all threads agree on control-flow decisions that affect execution of collectives. 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 [2] 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 [17].

[1] Throughout this paper, we highlight team operations in a bold, green color and collective operations in bold purple.

The work in this paper is in the context of the Titanium language, an explicitly parallel dialect of Java. Titanium uses the SPMD execution model and the partitioned global address space (PGAS) memory model; the latter allows a thread to directly access memory on any other thread, even if they do not physically share memory. Titanium's memory model is actually hierarchical, exposing three levels of memory hierarchy in the type system and compiler by distinguishing between thread-local, node-local, and global data.

2.1 The RSPMD Model

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*, in which the set of program threads can be divided into smaller subsets (teams) that cooperatively run pieces of code. MPI has communicators that allow teams of threads to perform collective operations. Similarly, the GASNet [5] runtime layer used in Titanium now has experimental support for teams and team collectives. Teams in MPI, UPC, and GASNet are non-hierarchical groupings of threads and do not place restrictions on the underlying thread structure of a team. A thread can be a part of multiple teams concurrently, making it easy to deadlock a program through improper use of teams. Even correct use of multiple teams can be difficult for programmers to understand and compilers to analyze, as they must reason about the order of team operations on each thread. Finally, teams in MPI, GASNet, and UPC do not have a hierarchical structure, so they cannot easily reflect the hierarchical organization of algorithms and machines.

Instead of the flat teams of MPI, GASNet, and UPC, we introduce the *recursive single program, multiple data (RSPMD)* programming model that uses hierarchies of teams. In this model, threads start out as part of a single, global team. This team can then be divided into multiple subteams, each of which can be recursively subdivided. Multiple, distinct hierarchies can be used in different parts of a program. Hierarchies can be created to match the underlying machine hierarchy, as in the Titanium memory model, or to match an algorithmic hierarchy, as in divide-and-conquer algorithms. At each point in the program, a thread is active in only a single team, and any collective operation that it invokes operates over that team. In §3, we take care to define RSPMD language extensions that enforce this restriction and prevent misuse of teams that would result in deadlock.

2.2 Related Work

While many current languages besides the SPMD languages mentioned above are locality-aware, only a handful of them incorporate hierarchical programming constructs beyond two levels of hierarchy.

In the Fortress language [1], 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. Hierarchically tiled arrays (HTAs) [3] allow data structures to be hierarchically decomposed to match a target machine's layout, which are then operated over in a data parallel manner. Other languages such as Chapel [7]

and the hierarchical place trees (HPT) [21] extension of X10 also have the concept of hierarchical locales. While these languages may be built on SPMD runtimes, they do not present the SPMD model of execution to the programmer.

Nested data parallelism allows hierarchical algorithms to be expressed in the context of data parallelism. The model has been implemented in NESL [4] and in Haskell [13]. However, irregular algorithms can be difficult to express in the data parallel model, and nested data parallel implementations have focused on vector and shared-memory machines rather than hierarchical machines. They also require more complicated compilers than SPMD languages.

The Sequoia project [9] 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. The *team parallel* model defined by Hardwick [11] is a data-parallel analogue of Sequoia, where threads are arranged into a hierarchy of teams, each of which is operated over in a data-parallel manner. Unlike RSPMD, this model does not allow the expression of explicit parallelism. In both Sequoia and Hardwick's model, communication is restricted to between parent and child tasks or teams, making the models unsuitable for many applications written in SPMD and PGAS languages.

The hierarchical single program, multiple data (HSPMD) model is in some sense the inverse of the RSPMD model. In RSPMD, an initial, fixed set of threads is recursively subdivided into smaller teams of cooperating threads. In HSPMD, on the other hand, there is only a single thread initially, and each thread can spawn a new set of cooperating threads. The Phalanx programming model uses a version of HSPMD [10].

3 RSPMD Language Extensions

In this section, we define language extensions for Titanium to implement the RSPMD model. In designing the new additions to the Titanium language, we had a few goals in mind for the extensions to satisfy: safety, flexibility, composability, and performance.

1. **Safety.** Team implementations in other SPMD 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 attempt to perform a collective operation on a different team; if the two teams overlap, then this situation results in deadlock. The Titanium team extensions should prevent such dependencies, as well as ensure that team collectives are textually aligned on all threads in the relevant team, as is done for existing global collectives.
2. **Flexibility.** Many applications make use of different thread groupings at different points in the program, such as a matrix-vector multiplication that requires both row and column teams. The team mechanism should be flexible enough to support such cases while still providing safety guarantees.
3. **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

mechanism should make it possible to interact with threads outside of an assigned team if necessary.

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.

3.1 Team Representation

To represent a team hierarchy, we introduce a new `Team` object, which represents a group of threads and contains references to parent and child teams, resulting in a hierarchy of teams. Like MPI or GASNet groups, `Team` objects specify team structures separately from their usage; this is useful when a program uses multiple different team structures or repeatedly uses the same structure, as in §4.2, 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 function `Ti.`**`defaultTeam`**() returns a special team that corresponds to the mapping of threads to the machine hierarchy, grouping together threads that share memory. The invocation `Ti.currentTeam`() returns the current team in which the calling thread is participating.

Figure 1(a) 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);
```

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`** method to divide it into three equally-sized subteams of four threads each. It then divides each of those subteams into two uneven, smaller teams. The **`splitTeamRelative`** call divides a team into subteams using IDs relative to the parent team. In this case, each child u of team `t` is split into two smaller teams, with threads 0, 2, and 1 of u assigned to the first subteam and thread 3 of u assigned to the second. This behavior allows the same code to be used to divide each of the three children of `t`, which would not be the case if **`splitTeamRelative`** used global IDs.

The `Team` class provides a few other ways of generating subteams, though we omit them for brevity. In addition, it includes numerous methods to query team properties; for example, the class provides a `myChildTeam` method for determining which child team contains the calling thread. Similarly, the `teamRank` method returns the rank of a team in its parent, which can be used to write code that is conditional on a team's rank.

3.2 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

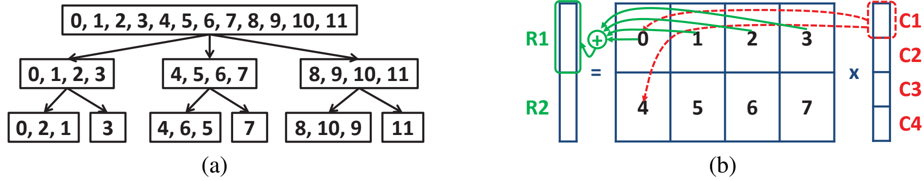

Fig. 1. Examples of (a) a team hierarchy and (b) blocked matrix-vector multiplication.

sets of threads that perform the same operation on different pieces of data. We introduce 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 structure 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 of threads 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 a thread exits a partition, it rejoins its previous team.

As a concrete example, consider a climate application that uses the team structure in Figure 1(a) 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.

Consider a parallel matrix-vector multiplication as in Figure 1(b), where the matrix is divided in both dimensions among 8 threads, with four thread columns and two rows. 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 with its current team set to the thread's subteam specified in T, so that thread ranks and collective operations in B are with respect to that subteam. As mentioned above, each subteam also has a rank, which can be used to determine the set of data that the subteam is to operate on.

As an example, the following code executes reductions across the rows of a matrix:

```
teamsplit(t) {
  Reduce.add(data[t.myChildTeam().rank()], myData);
}
```

The reduction executes over the current team inside the teamsplit on each thread, which is its associated child team of `t`. As a result, data from threads 0 to 3 are reduced to produce a result for team 0, and data from threads 4 to 7 are combined into a result for team 1.

It may be apparent that the partition statement can be implemented in terms of teamsplit, with teams executing code based on their ranks. While this is true, we decided that separate constructs for task and data decomposition would result in cleaner and more readable code than a single construct combined with branching.

Common Features. Both the partition and teamsplit constructs are dynamically 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 and have child teams of its own). 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 the merge sort in §4.2.

In order to meet the composability design goal, 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. Both the partition and the teamsplit construct are also considered collective operations, so they must be textually aligned in the program. The combination of the requirement that all threads must agree on the set of subteams when entering a partition or teamsplit construct, dynamic scoping of the constructs, and textual collective alignment ensures that no circular dependencies exist between different collective operations. In the next section, we describe how textual collective alignment is enforced.

3.3 Alignment of Collectives

With the introduction of hierarchical teams, alignment of collectives must be checked dynamically at runtime. The full details of dynamic alignment checking are described elsewhere [17,18,14], but we will summarize the main ideas here.

Enforcement of collective alignment is divided into two phases, a local tracking phase and a collective checking phase. In the tracking phase, each thread records the control flow decisions that it makes that may affect execution of a collective. The Titanium compiler already statically computes which conditionals may do so; such conditionals are a small subset of all conditionals in a program, so the cost of tracking is low. Memory usage and communication costs can be minimized by computing a running hash of all such control flow decisions.

The checking phase occurs when a thread reaches a collective operation. Prior to entering the collective, it waits for a broadcast of the alignment hash from thread 0 in its current team. Once it receives thread 0's hash, it compares it to its own and generates an error if the two hashes do not match. Otherwise it proceeds with the collective operation. If no thread generates an error, then all agree on the hash, implying that they also agree on all control flow decisions that affect the collective operation, guaranteeing textual alignment.

Dynamic alignment checking avoids deadlock by requiring that every collective operation be preceded by an alignment check. This check itself executes a collective broadcast over a thread's current team, but this collective is the same on all the threads in the team, so it will never deadlock as long as a check is also performed when changing team contexts.

Previous work has demonstrated that the cost of dynamic alignment tracking and checking is negligible in actual programs [17]. In addition, an optional debugging mode for alignment checking is provided, in which the control flow history is compared between two threads whose hashes mismatch, and the earliest mismatch is reported. This mode also does not measurably degrade performance. Thus, not only is deadlock avoided with low overhead, but a meaningful error is generated that directs the programmer to the source of the error. This may be far from the point of detection, so alignment checking can facilitate debugging.

4 Application Case Studies

We now present case studies of four applications we used to guide the design of the RSPMD language extensions and evaluate their effectiveness: conjugate gradient, parallel sort, particle in cell, and stencil.

4.1 Test Platforms

We tested application performance on two machines, a Cray XE6 and an IBM iDataPlex, both located at the National Energy Research Scientific Computing Center (NERSC) at the Lawrence Berkeley National Laboratory (Berkeley Lab). The Cray XE6, called *Hopper*, consists of two twelve-core AMD MagnyCours 2.1 GHz processors per node, each of which consists of two six-core dies. Each die is referred to as a *non-uniform memory access (NUMA) node*, since each die has fast access to its own memory banks but slower access to the other banks. The IBM iDataPlex system, known as *Carver*, is a cluster of eight-core, 2.67 GHz Intel Nehalem processors connected by a 4X QDR InfiniBand network. Memory considerations limited us to 32 nodes for most benchmarks and prevented larger problem sizes from being run on the IBM machine.

In most of the benchmark applications, we focused on optimizing distributed performance. As a result, we used performance on a single shared-memory node or NUMA node as the baseline for our experiments. Optimizing execution solely on shared-memory multicores is beyond the scope of this paper.

4.2 Algorithmic Hierarchy

We began by examining two algorithms that are difficult to express in the flat SPMD model: conjugate gradient and merge sort.

Conjugate Gradient. The conjugate gradient (CG) application is one of the NAS parallel benchmarks [2]. 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, as described previously in §3.2. In addition to the reductions mentioned there, in each iteration of the algorithm, the elements of the source vector must be distributed to the threads that own a portion of the corresponding matrix column. Thus, team collective operations are required over both rows and columns of threads.

Prior to our language extensions, Titanium only supported collectives over all threads in a program. Thus, the original Titanium implementation of CG [8] required hand-written reductions over subsets of threads. These reductions required extensive development effort to implement, test, and optimize.

The team implementation of CG, on the other hand, makes use of both row and column teams. The existing CG code already computes the row and column number of each thread; we use them to divide the threads into row teams with a call to `splitTeamAll`(), which takes in the child team number and rank for the calling thread as arguments. We then use `makeTransposeTeam`(), which swaps the child team number and rank for each thread, to create column teams from row teams.

```
rowTeam = new Team();
rowTeam.splitTeamAll(rowPos, colPos);
columnTeam = rowTeam.makeTransposeTeam();
```

We use all-to-one reductions across each row team to send the result of that row team to a single thread in the team. We then use a broadcast to send data from that thread to all threads in the same column. Each reduction or broadcast requires only a single library call, as shown below.

```
teamsplit(rowTeam) { // Reduce row results to one thread.
  Reduce.add(allResults, myResults, rowTarget);
}
... // Perform required copies across columns.
teamsplit(columnTeam) { // Broadcast from column source.
  myOut.vbroadcast(columnSource);
}
```

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 of creating teams is amortized over all iterations of the algorithm.

Figure 2(a) compares the performance of the team-based version of CG to the original hand-rolled implementation on a Cray XE6 and an IBM iDataPlex. We show strong scaling (fixed problem size) results for the Class B problem size. (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 all-to-all reductions with optimized GASNet all-to-one reductions and broadcasts improves performance over the original version. We achieve speedups over the original code of 1.6x for Class B at 128 threads on the XE6. On the IBM iDataPlex, Class B only scales until 64 threads, at which point the team version is 2.1x as fast as the original code.

We also ran experiments using the Class D problem size, though the graph is omitted for brevity. On the XE6, the team-based version achieves a speedup of 1.5x over the original code at 1024 threads. On the IBM machine, Class D achieves a speedup of 1.6x at 256 threads, at which point the original version stops scaling, and 2.7x at 512 threads.

Shared-Memory Merge Sort. Merge sort is a canonical example of a divide-and-conquer algorithm. An initial set of keys is recursively divided in half, until some threshold is reached. The subsets are sorted individually and then recursively merged with each other until all keys are in a single sorted set. This algorithm can be parallelized on a shared-memory machine by forking a new thread each time a set of keys is divided in two. However, since the SPMD model does not allow new threads to be created, merge sort is difficult to express in the flat SPMD model.

In the RSPMD model, however, merge sort is easily expressible by starting with a team of all threads and then recursively dividing both the set of keys and the team until only a single thread remains. Then each thread sequentially sorts its keys, and the sorted subsets are merged in parallel by assigning each merge operation to one thread from the subsets that are to be merged.

In order to express merge sort in this way, a team hierarchy is constructed that consists of a binary tree, in which each node contains half the threads of its parent. The

following code constructs such a hierarchy, using the **splitTeam** library method 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 keys, 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. (The sequential sort and merge functions are omitted for brevity.)

```
static 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)
      allRes[myProc] = merge(myRes(), otherRes(), newRes())
  }
}
```

As illustrated in the code above, 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 from the Java standard library) and took just two hours to write and test. This sort is used as part of the larger distributed sort implementation below, so we will defer performance results until then.

4.3 Machine Hierarchy

We now turn our attention to optimizing algorithms for hierarchical machines. We examined three algorithms: distributed sort, stencil, and particle in cell.

Distributed Sort. The first algorithm we examined for hierarchical optimizations was distributed sorting, specifically the sample sort algorithm [12] on 32-bit integers. This algorithm consists of two phases: an initial phase that computes pivots based on a sample of all the keys and then redistributes the keys among all threads according to the pivots, and a second phase that sorts keys locally.

We explored three different versions of sample sort. The first is a *flat* version that is purely distributed, ignoring the hierarchical structure of the machine. This version uses sample sort across all threads and sequential sorting on each individual thread. In this version, key redistribution requires $n(n-1)$ messages where n is the total number of threads. The second is a *composed* version that uses sample sort across nodes rather than threads, but then uses shared-memory merge sort on each node. Here, key redistribution

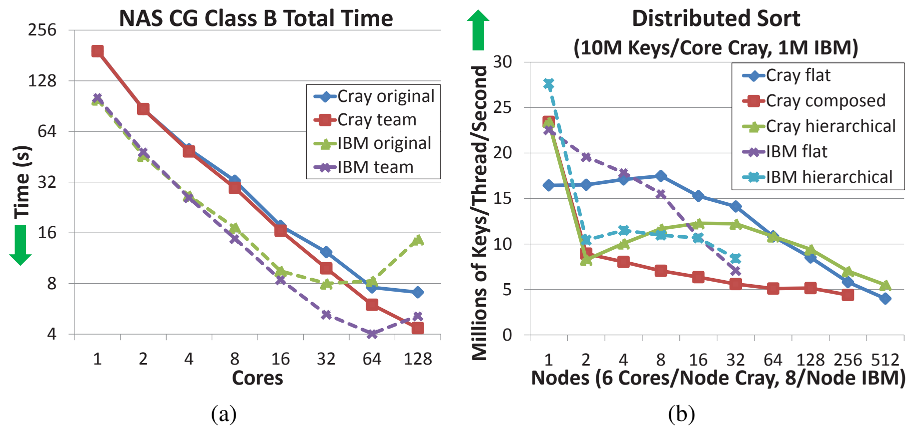

Fig. 2. (a) Strong scaling performance of conjugate gradient; (b) distributed sort performance, with a constant number of keys per thread.

requires $m(m-1)$ messages, where m is the number of nodes. However, the composed version uses only a single thread per node for sampling and redistribution, so that it is equivalent to composing a communication library such as MPI with a shared-memory library such as Pthreads. The final version is *hierarchical*; it improves on the composed version by using all available parallelism in the sample and redistribution phase. The RSPMD model enable this version to be expressed, since it exposes hierarchy in the context of a single model.

The composed version, though it does not take full advantage of the hierarchy exposed by RSPMD, does illustrate its composability features. The following is the code required to implement the composed version, where `sampleSort` is the sampling and redistribution code from the flat sample sort:

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

The RSPMD team constructs make this algorithm trivial to implement, requiring only a few 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 sampling and redistribution on one of those subteams, after which the node teams execute the shared-memory sort. The team hierarchies in `sampleSort()` and in the shared-memory sort compose cleanly, without any modifications required.

Figure 2(b) compares the number of keys sorted per thread per second in the different versions of distributed sort. On both machines, the hierarchical version scales better than the flat version, resulting in a speedup of 1.4x for the hierarchical version

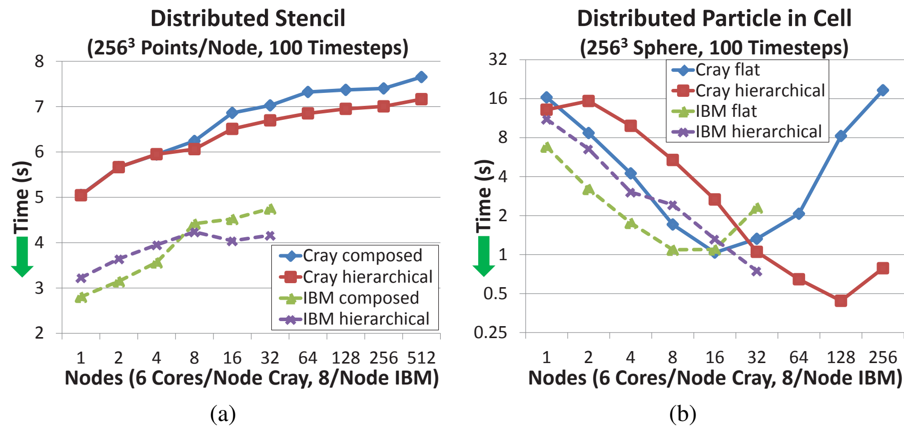

Fig. 3. (a) Weak scaling performance of stencil; (b) strong scaling performance of particle in cell.

over the flat version on 512 NUMA nodes (3072 cores) of the XE6 and 1.2x on 32 nodes (256 cores) of the IBM iDataPlex. Since sorting in general is not a linear time algorithm, the decrease in efficiency shown in Figure 2(b) at higher numbers of threads is not unexpected.

As can be seen in Figure 2(b), the composed version performs significantly worse than the flat and hierarchical versions on the Cray machine. Since the composed version is equivalent to composing distributed and shared-memory libraries, this demonstrates the importance of exposing hierarchy within a single model to obtain optimal performance.

Stencil. As another example of comparing the composition of distributed and shared-memory code to a true hierarchical version, we examined a stencil benchmark. A *stencil* is a nearest-neighbor computation over a structured n-dimensional grid and consists of multiple iterations in which the value of each grid point is updated as a function of its previous value and those of its neighboring points. In this benchmark, we execute a seven-point stencil over a three-dimensional grid. Since we are primarily concerned with optimizing distributed communication, we use a naïve, untuned shared-memory version of stencil as part of our experiments.

We compared two implementations of distributed stencil. As with distributed sort, the composed version uses a single Titanium thread per node to perform communication and multiple threads per node to perform computation in the external library. We also wrote a hierarchical version that uses multiple threads for both communication and computation. Figure 3(a) shows weak scaling (constant problem size per thread) performance of the stencil variants. On both machines, the hierarchical version outperforms the composed variant at higher node counts, improving performance by up to 7% on the Cray machine and 14% on the iDataPlex.

Particle in Cell. The final benchmark we examined was *particle in cell*, which models the communication pattern in one phase of a heart simulation written in Titanium [22].

In this phase, a set of particles interact with an underlying three-dimensional fluid grid. We model this interaction by updating each fluid cell with a value from each of the particles that the cell contains. Both particles and the fluid grid are divided among the threads; however, a thread's particles are not generally located in its portion of the fluid, requiring communication to perform updates.

We compared two versions of particle in cell. The *flat* version divides the fluid grid and particles between each thread, which separately process their fluid cells and particles, performing any required communication directly between different threads. In the *hierarchical* version, the fluid and particles are divided among nodes, and the threads in a node cooperatively process the node's fluid cells and particles. In this version, communication is aggregated between nodes.

Figure 3(b) compares the performance of the two versions of particle in cell on a 256^3 fluid grid with particles on the surface of a sphere. The flat algorithm does not scale beyond 16 nodes on the Cray machine and 8 nodes on the IBM machine, while the hierarchical algorithm scales up to 128 and 32 nodes, respectively. On the other hand, the flat algorithm performs about twice as fast as the hierarchical version up to the former's scaling limits. This is largely due to the fact that the Titanium and GASNet runtimes are not optimized for shared memory. As a result, though the hierarchical algorithm does scale more, it requires four times as many processors to improve running time beyond the best performance of the flat algorithm.

5 Conclusion

In this paper, we presented RSPMD, an extension of SPMD that enables hierarchical programming in an explicitly parallel model. We designed RSPMD extensions to the Titanium language, combining a team data structure and dynamically scoped usage constructs to prevent erroneous usage of teams. We also described how to enforce textual alignment of team collectives at runtime, further avoiding errors in using team collectives.

We implemented four benchmarks using the RSPMD model: conjugate gradient, sorting, stencil, and particle in cell. We demonstrated that hierarchical teams enable divide-and-conquer algorithms such as sorting to be implemented elegantly, and that team collectives provide better performance and expressiveness than hand-written alternatives in conjugate gradient. We also demonstrated that hierarchical teams enable optimizations for hierarchical machines to be written in the context of a single programming model, enabling increased performance in sorting and better scaling in particle in cell. We further showed that our hierarchical model beats the standard mechanism of combining a distributed library with a shared-memory library in both sorting and stencil. These results demonstrate that the RSPMD model provides significant expressiveness and performance advantages over the flat SPMD model.

References

1. E. Allen et al. *The Fortress Language Specification, Version 0.866*. Sun Microsystem Inc., 2006.

2. D. Bailey et al. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3), 1991.
3. G. Bikshandi et al. 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*, 2006.
4. G. E. Blelloch. NESL: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.
5. D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, 2002.
6. W. Carlson et al. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
7. Cray Inc. *Chapel Specification 0.4*, 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. K. Fatahalian et al. Sequoia: Programming the memory hierarchy. In *Supercomputing, 2006. SC '06. Proceedings of the ACM/IEEE SC 2006 Conference*, 2006.
10. M. Garland, M. Kudlur, and Y. Zheng. Designing a unified programming model for heterogeneous machines. In *Supercomputing 2012*, 2012.
11. J. C. Hardwick. *Practical Parallel Divide-and-Conquer Algorithms*. PhD thesis, Carnegie Mellon University, 1997.
12. J. Huang and Y. Chow. Parallel sorting and data partitioning by sampling. In *7th International Computer Software and Applications Conference*, 1983.
13. S. P. Jones et al. Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, 2008.
14. A. Kamil. *Single Program, Multiple Data Programming for Hierarchical Computations*. PhD thesis, University of California, Berkeley, 2012.
15. 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*, 2005.
16. A. Kamil and K. Yelick. Hierarchical pointer analysis for distributed programs. In *The 14th International Static Analysis Symposium (SAS 2007), Kongens Lyngby*, 2007.
17. 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*, 2009.
18. A. Kamil and K. Yelick. Hierarchical additions to the SPMD programming model. Technical Report UCB/EECS-2012-20, University of California, Berkeley, 2012.
19. Message Passing Interface Forum. MPI: A message-passing interface standard, version 1.1, 1995.
20. R. Numrich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.
21. Y. Yan et al. 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*, 2009.
22. S. M. Yau. Experience in using Titanium for simulation of immersed boundary biological systems. Master's thesis, University of California, Berkeley, 2002.
23. K. Yelick et al. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing*, 1998.