A Team Analysis Proposal for Recursive Single Program, Multiple Data Programs

Amir Ashraf Kamil

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-183
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-183.html

August 6, 2012

Copyright © 2012, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A Team Analysis Proposal for Recursive Single Program, Multiple Data Programs

Amir Kamil
Computer Science Division, University of California, Berkeley
kamil@cs.berkeley.edu

August 6, 2012

1 Introduction

Large-scale parallel machines are programmed primarily 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. The SPMD model's simplicity enables a variety of program analyses, including data locality analysis [9], data sharing analysis [10], concurrency analysis [6], and pointer analysis [7]. However, the flat SPMD model does not fit well with divide-and-conquer parallelism or hierarchical machines that mix shared and distributed memory.

In order to facilitate expression of hierarchical algorithms, we introduced the recursive single program, multiple data (RSPMD) model [8]. As in the flat SPMD model, a fixed set of threads exist throughout program execution. However, this set of threads can be recursively subdivided into smaller groups of cooperating threads, or *teams*. Collective operations can be executed over teams, allowing teams to perform different tasks or the same task over different sets of data. We demonstrated that the RSPMD model enables productive expression of high performance hierarchical algorithms. We did not, however, address the analyzability of RSPMD programs.

In this report, we lay the groundwork for a *team analysis* to infer properties of the teams in an RSPMD program, specifically in the context of the Titanium language [14]. Such an analysis is necessary for many client analyses, including pointer and concurrency analysis. We discuss the information required by those analyses and provide a rough sketch of a team analysis for computing that information. We hope that this sketch can be used as a basis for future implementation of a complete team analysis.

2 Background

The recursive single program, multiple data (RSPMD) model of parallelism consists of a fixed set of threads that can be recursively subdivided into smaller teams of cooperating threads. In this section, we describe the RSPMD language extensions to Titanium that are relevant to team analysis. We also provide background on pointer and concurrency analysis, two potential clients for team analysis.

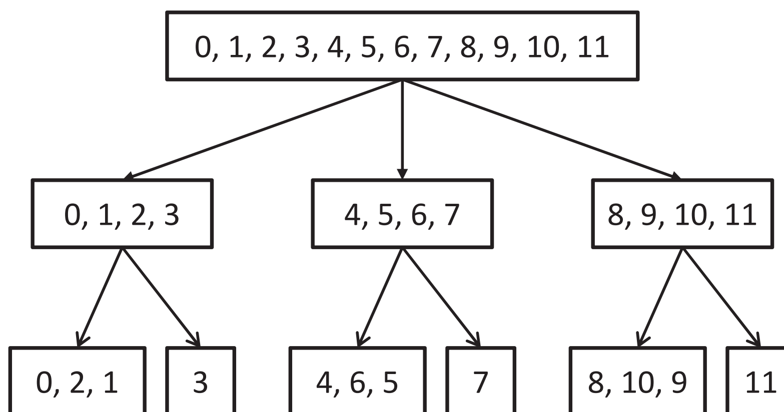

Figure 1: An example of a team hierarchy.

2.1 Language Extensions for Teams

The Titanium implementation of RSPMD [8] is divided into two pieces: team representation and team usage. The former consists of the data structures that represent teams, as well as the interfaces for constructing teams. The latter consists of language constructs to change team contexts.

2.1.1 Team Representation

The team data structure is implemented by the `ti.lang.Team` class, representing threads arranged in arbitrary hierarchies. Figure 1 shows such a hierarchy. Each node in the hierarchy corresponds to a `Team` object. The top level always consists of all the threads in the program. Each `Team` can have any number of children, and the children of a non-leaf `Team` must consist of non-overlapping sets of threads whose union is the set of threads in the parent.

The team library provides multiple ways of creating teams. The call `Ti.defaultTeam()` creates a team hierarchy whose structure reflects that of the underlying machine hierarchy at runtime. The call `new Team()` with no arguments creates a shallow copy of the current `Team`, and the call `new Team(t)` creates a shallow copy of the `Team t`. Various **`splitTeam`** methods create child `Team`s when invoked on a leaf `Team`.

As an example, Figure 1 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);
```

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 the IDs used in **`splitTeamRelative`** are relative to the parent team, so the same code can be used to divide each of the three teams at the second level.

The exact details of the various **`splitTeam`** methods are not relevant to team analysis, so we ignore them here. The only relevant effect of all such methods is that they divide a parent team into child teams.

2.1.2 Team Usage

All Titanium code runs in the context of a particular team, meaning that at any point in execution, each thread is executing as a member of exactly one team, represented by a `Team` object. The Titanium runtime keeps track of the current `Team` on each thread. Collective operations such as barriers and broadcasts operate over the current team.

In order to change team contexts, programmers must make use of either the partition or the teamsplit construct. The *partition* construct divides threads into subteams that execute distinct tasks:

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

A `Team` object, which must match 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. 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 1 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.

The *teamsplit* construct divides threads into subteams that execute the same task:

teamsplit(T) B

The parameter T must be a `Team` object, which must match the current team at the top level. The construct causes each thread to execute block B as a member of its new subteam specified by the team argument.

Entering a partition or teamsplit construct walks one level down a team hierarchy, and exiting a construct moves one level up. The Titanium runtime sets the current `Team` object to the appropriate team when entering or exiting a partition or teamsplit. The constructs can be nested either lexically or through function calls, allowing threads to walk multiple levels down a team hierarchy. Recursion can be used to traverse hierarchies of arbitrary depth, as in the following:

```
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. Such code can be used to implement divide-and-conquer algorithms.

2.2 Pointer Analysis

Pointer analysis, or *points-to analysis*, statically determines the set of objects that may be referenced by each pointer in a program. Pointer analysis was first described and implemented for the C programming language by Emami [4] and

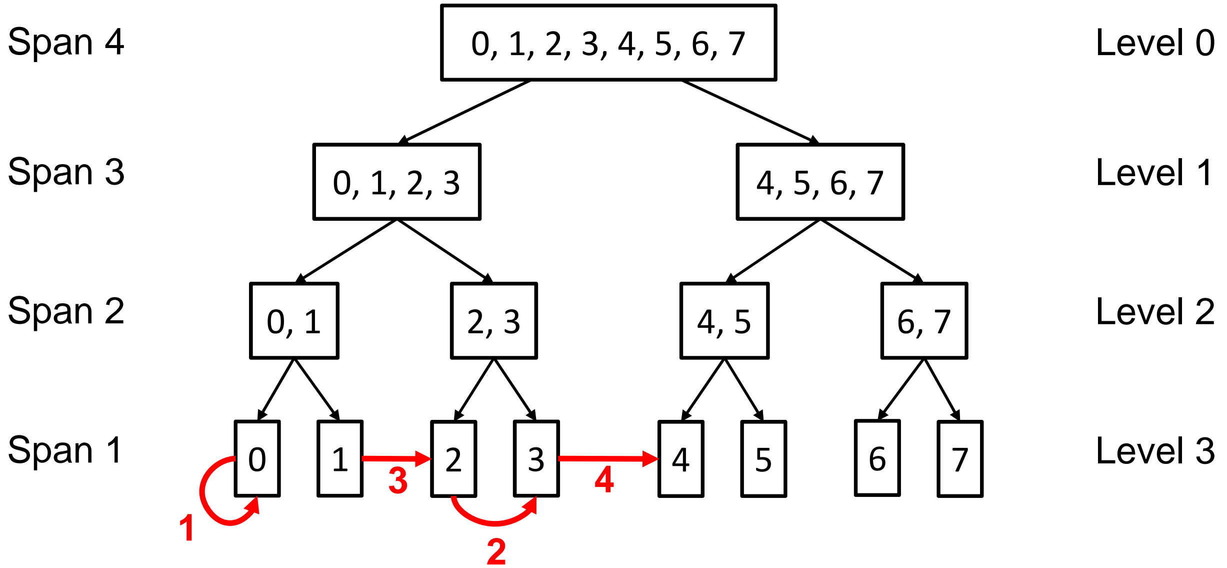

Figure 2: An example of a machine hierarchy and pointer span.

Andersen [1]. Here, we provide an overview of context-insensitive, flow-insensitive pointer analysis for object-oriented languages such as Java and Titanium.

Pointer analysis is an example of *abstract interpretation*, where the execution of a program is approximated statically. In the case of pointer analysis, each allocation site corresponds to an *abstract location*, which represents all objects allocated at that site at runtime. Each allocation site only constructs objects of a single type[1], and its corresponding abstract location has the fields specified by that type. Every variable in the program and field of each abstract location has a *points-to* set consisting of abstract locations that may be referenced at runtime by that variable or field. The analysis iterates over the whole program, updating points-to sets according to statements and expressions in the program. For example, consider the assignment

```
v = x.f;
```

The analysis examines the points-to set of `x`, and for each abstract location in that set, it examines the points-to set for the field `f`, collecting the resulting abstract locations. These locations then are copied into the points-to set of `v`. The analysis continues to iterate over the program, interpreting each statement and expression, until a fixed point is reached.

We have previously described a pointer analysis for Titanium that takes into account machine hierarchy [7]. For a machine hierarchy with height h, the *span* of a pointer is an integer in the range $[1, h]$ and denotes the threads on which the referenced data may reside. A span of 1 implies that the pointer target must be on the same thread as the pointer itself, a span of h allows the target to be on any thread, and spans strictly between 1 and h allow the target to be on a subset of threads within the corresponding distance of the source thread in the machine hierarchy. Figure 2 shows examples of pointer span.

In the hierarchical pointer analysis, abstract locations have both a corresponding allocation site and a span. For each allocation site, h abstract locations are created, each with a separate span. When an abstract location escapes

[1] We ignore Java reflection for simplicity.

its creating thread, either through a collective operation such as a broadcast or through a dereference, the result is an abstract location with the same allocation site but a greater span. As a concrete example, consider the following statement:

```
w = broadcast x from 0;
```

This statement assigns thread 0's value of `x` to `w` on all threads. Suppose that in the pointer analysis, the points-to set of `x` contains the lone abstract location $(l, 1)$, meaning the abstract location with span 1 corresponding to allocation site l. Then the broadcast results in the abstract location (l, h), which is added to the points-to set of `w`. The resulting span is h since the broadcast is a global operation over all threads.

2.2.1 Applications

Many client analyses and optimizations can take advantage of the information computed by pointer analysis. Three examples are locality analysis, sharing analysis, and race detection.

Locality Analysis In partitioned global address space (PGAS) languages such as Titanium, a pointer can reference data located on any thread, even if the source and target of a pointer do not share the same physical memory space. A pointer has a particular span that specifies where the referenced data may be located, with greater spans allowing reference to data on more distant threads. In Titanium, pointers have maximal span by default and must be qualified as **`local`** in order to specify a smaller span. Unfortunately, pointers with greater span can be more expensive to dereference than pointers with smaller span, even when the target location is the same. Liblit and Aiken demonstrated up to a 56% improvement in application running time with an automated analysis to infer local qualifiers [9]. This illustrates the need for a *locality analysis* to compute locality information at compile-time.

Liblit and Aiken's analysis uses a constraint solver in order to compute locality information. Unlike pointer analysis, their analysis does not distinguish between allocation sites, so pointer analysis should be able to produce more precise results. Computing locality information from points-to sets is trivial in a hierarchical pointer analysis: the span of a variable is the maximum span of any of the abstract locations in its points-to set.

Sharing Analysis In parallel programs, *sharing analysis*, also called *escape analysis*, is relevant to many applications. Sharing analysis statically determines what data may be shared between multiple threads. If an object is provably private to a single thread, a compiler can safely allocate it in private memory, such as the thread's stack or scratchpad memory. Synchronization can also be eliminated for private objects. This is especially important in library code that tends to be over-synchronized for safety. For example, the `java.util.Vector` is synchronized, so an application that uses thread-private instances of the class can execute many spurious synchronizations. Bogda and Hölzle demonstrated up to a 36% improvement in benchmark performance by automatically eliminating synchronization on private objects in Java applications [2]..

Hierarchical pointer analysis can be used for sharing analysis. An allocation site is private if no points-to set contains an abstract location corresponding to that site with span greater than 1. A variable is private if all abstract locations in its points-to set correspond to private allocation sites.

Race Detection A *race condition* occurs when two memory accesses can occur concurrently on different threads, they can be to the same memory location, and at least one of them is a write [11]. Race conditions can result in erroneous program behavior, so static detection of races is an important problem in program analysis.

Pointer analysis is a key component of race detection, since it can determine whether or not two memory accesses may be to the same location. Variables are said to be *aliased* if they may reference the same location. In hierarchical pointer analysis two variables `x` and `y` are *aliased across threads* if:

- The points-to set of x contains an abstract location (l, a) and that of y contains a location (l, b) that correspond the the same allocation site.
- Either a or b is greater than 1, meaning that at least one the referenced locations may not be thread-local and thus may overlap with the other.

As an example, consider the following code:

```
1 Foo w = new Foo();
2 w.f = 4;
3 Foo z = broadcast w from 0;
4 return z.f;
```

The points-to set of w contains the abstract location $(1, 1)$, so that of z contains the location $(1, h)$, where h is the height of the machine hierarchy. Thus w and z are aliased across threads, and z is aliased with itself across threads. On the other hand, w does not alias itself across threads. As a result, the write on line 2 does not by itself constitute a race condition, since the writes are to thread-local locations on all threads. However, the combination of that write and the read on line 4 is a race condition, since other threads read the location in line 4 that is concurrently written in line 2.

2.3 Concurrency Analysis

Information concerning which statements and expressions may run concurrently is important in many analyses and optimizations. *Concurrency analysis* statically computes this information, analyzing the synchronization structure of a parallel program. We previously described a precise and efficient concurrency analysis for SPMD programs [6]. This analysis traverses a graph representation of a program. Two expressions are never concurrent if all paths between them contain a global synchronization operation.

As an example of concurrency analysis, consider the following code that makes use of a barrier operation:

```
1 Foo w = new Foo();
2 w.f = 4;
3 Foo z = broadcast w from 0;
4 Ti.barrier();
5 return z.f;
```

A *barrier* operation requires all threads to reach it before any can proceed. In the above code, the only path from the write on line 2 to the read on line 5 contains a global barrier synchronization. As a result, they never execute concurrently, and the code does not contain a race condition.

Concurrency information is useful for analyses and optimizations besides race detection. For example, it can be used to enforce a sequentially consistent memory model [5]. It can also be useful in synchronization elimination, particularly in eliminating redundant barrier operations [3, 12, 13].

2.4 Analysis over Teams

With the addition of teams, collective operations need not be global. As a result, widening abstract locations to the maximum span in pointer analysis may be overly imprecise. For example, a broadcast over a lower level in `Ti.defaultTeam()`, the team that matches the machine hierarchy, should produce an abstract location with span smaller than h. Thus, team-awareness can improve locality analysis.

In addition to span with respect to the machine hierarchy, it would be useful in pointer analysis to compute spans with respect to arbitrary team hierarchies. Similarly, concurrency analysis can compute team-level concurrency information in addition to the global information it computes now. As an example, consider the following code:

```
1 Foo y;
2 teamsplit(t1) {
3   Foo x = new Foo();
4   x.f = 4;
5   y = broadcast x from 0;
6 }
7 ...
8 teamsplit(t2) {
9   Ti.barrier();
10  return y.f;
11 }
```

If `t1` and `t2` reference the same concrete team at runtime, say T_a, then the write on line 4 does not conflict with the read on line 10. This is due to the fact that the threads that read the location `y.f` and the thread that writes to it belong to the same child team of T_a, so `x` and `y` are aliased across the child team of T_a but not globally. However, all threads in that child team synchronize on the barrier in line 9, so the two accesses are non-concurrent in the child team of T_a, and no race condition exists. On the other hand, if `t1` and `t2` do not refer to the same team, then a race condition may result. Thus, a team analysis is required that can statically determine what teams may be active at any program location.

3 Team Analysis

As discussed in §2.4, team-aware versions of both pointer and concurrency analysis require precise information concerning team hierarchies. A new team analysis will compute this information.

Team analysis is not concerned with the actual layout of threads in a team; rather, it computes the set of all teams in the program, how they are related to each other, and the locality of each team. It further tags every expression in the program with the set of teams under which the expression may execute.

We sketch out a team analysis based on Andersen's context-insensitive, flow-insensitive points-to analysis [1]. This sketch is incomplete; we do not provide actual abstract semantics rules and do not define them or the abstract domain in any detail. Rather, the sketch is intended to be a framework from which a complete team analysis can be defined.

3.1 Analysis Requirements

Team-aware pointer and concurrency analyses rely on the set of teams and their uses computed by team analysis. As a result, the precision and correctness of those analyses is dependent on the precision and correctness of team analysis. Consider the following example:

```
1 Team a = new Team();
2 a.splitTeam(rand.nextInt(Ti.numProcs()));
```

Even if the above code is executed in the context of a single team, if it is executed multiple times, then the subteams constructed in the second line may differ in each execution. Team analysis, however, will represent all such concrete teams as a single abstract team. As a result, a collective that operates on that abstract team may at runtime execute on different concrete teams at different times, implying that it does not prevent code before and after it from being concurrent in any of those concrete teams. Thus, concurrency analysis must be aware that the abstract team may represent multiple concrete teams in order to compute correct results. As such, team analysis must conservatively determine which abstract teams represent multiple concrete teams.

In addition, it is possible for an abstract team to represent multiple levels in a concrete team, such as when a team is constructed recursively. Team analysis must determine and report when this is the case.

3.2 Abstract Domain

The abstract domain of a typical pointer analysis includes abstract locations, each of which is the abstraction of an allocation site in the program, and points-to sets for each variable in the program, which contain the set of abstract locations whose concretization can be referenced by the variable at runtime. In an object-oriented language, each abstract location has a type and points-to sets for each reference field specified by that type[2], representing the locations that the fields of the corresponding concrete locations can refer to. Arrays also have points-to sets, which can be a single set for all indices in an array or multiple sets for different indices or sets of indices.

In addition to the typical abstract domain in a pointer analysis, the abstract domain in team analysis includes abstract teams. An *abstract team* is the abstraction of a single level in a team hierarchy, though its concretization may be multiple levels in multiple team hierarchies. Each abstract team t has a number of attributes:

- $creator(t)$: the program location at which t is created
- $canonical(t)$: the canonical abstract team that is equivalent to t; may be this t itself
- $parent(t)$: a parent abstract team, if any
- $children(t)$: a set of zero or more child abstract teams
- $multiteam(t)$: whether or not the team may correspond to multiple concrete teams; note that a child team is never created for a team t such that $multiteam(t)$ is true, so that either $multiteam(u)$ is false for any ancestor u of t, or t is equivalent to the global team and has no ancestors
- $multilevel(t)$: whether or not the team may correspond to multiple levels in the same concrete team; note that if $multiteam(t)$ is true, then $multilevel(t)$ must be true as well, since the analysis does not create children for teams t for which $multiteam(t)$ is true
- $locality(t)$: the level in the machine hierarchy in which all threads in the same concrete subteams corresponding to this abstract team reside in the same machine-level team
- $level(t)$: the team's level from the root in its hierarchy, as reflected by following the parent pointers to the root

Each operation in the program has a *team set*, the set of abstract teams under which the operation can execute at runtime. Each method in the program also has a team set, which contains the abstract teams under which the method can execute. Finally, each abstract team location has an associated team set as well.

3.3 Abstract Semantics

Inference rules for team analysis include those in standard pointer analysis for determining the set of abstract locations and the points-to set of each location in the program. To these we add rules to deal with abstract teams and team sets.

Initially, the only abstract teams in the analysis are those corresponding to each level in the machine hierarchy team. All team sets are initialized as empty, with the exception of that of the `main` method and the initial statement of each static initializer in the program[3], which are initialized to contain the global team.

[2] In Java, objects may be allocated by reflection and thus have unknown static type. In such a case, the analysis must treat the corresponding abstract location's fields conservatively.

[3] We assume that each class's static initalization expressions are collected into a single static intializer block for the class.

The following is an overview of the team analysis rules for various constructs in the Titanium language:

- The team set for a method body is the same as the team set of the method.
- A sequencing operation copies the team set of the first expression or statement to the team set of the second.
- A method call copies the team set of the call to the team sets of the possible target methods.
- A team allocation expression (i.e. **new** `Team()`) has an abstract team t associated with the corresponding abstract location. If such a team doesn't exist, then one is created. If the allocation expression's team set S contains a single team or multiple teams with the same canonical team u, then u's attributes are copied to t. If S contains multiple distinct teams, however, then t's attributes are set as follows:
  - $creator(t)$ is set to the allocation expression
  - $canonical(t)$ is set to t, except as indicated in the following rules
  - $multiteam(t)$ is set to true
  - $multilevel(t)$ is set to true
  - $locality(t)$ is set to the least upper bound of $locality(u)$ over all teams u in S
  - if $parent(u)$ is the same for all teams u in S, then $parent(t)$ is set to that team; otherwise, $canonical(t)$ is set to the least upper bound over all teams u in S, $multilevel(t)$ is set to true, and $parent(t)$ is set to $parent(canonical(t))$
  - $level(u)$ is set to one more than $level(parent(t))$, where $level(null)$ is defined as -1
  - $children(t)$ is set to empty, and all former descendants are marked for deletion and will be replaced by t in all team sets in subsequent iterations of the analysis

  Note that the abstract location for each allocation expression corresponds to a single abstract team. Each time an allocation expression is analyzed, the above rules are applied to that abstract team, and its attributes are updated if needed. The restriction of one abstract team per allocation expression is necessary to ensure that the set of abstract teams is finite and that team analysis terminates. It also prevents multiple levels in the same team hierarchy from having the same allocation expression as creator.

- An expression `t.`**`splitTeam`**`()` is assigned a new abstract location cl for each abstract location pl in the points-to set P of `t`, and cl is added to the points-to set corresponding to the child teams of pl. Now the parent abstract location pl has an associated abstract team set S. For each abstract team pt in S, if the creator of pt or any of its ancestors is the same **`splitTeam`**, then $multilevel(pt)$ is set to true and pt is added to the team set of cl. Otherwise, if the creator of one of pt's children is the **`splitTeam`**, then that child is added to the team set of cl. Otherwise, if $multiteam(pt)$ is true, then pt is added to cl's team set. Otherwise, a new child abstract team ct is constructed and added to $children(pt)$ as well as the team set for the abstract location cl. Then ct's attributes are set as follows:
  - $creator(t)$ is set to the **`splitTeam`**
  - If $canonical(pt)$ is equal to $canonical(pt')$ for a previously examined team pt' in S and the arguments of the **`splitTeam`** are constant, then $canonical(ct)$ is set to the newly created child team ct' of pt'. Otherwise $canonical(ct)$ is set to ct.

- If the arguments of the `splitTeam` are non-constant and the call may be executed more than once in a program run, then $multiteam(ct)$ and $multilevel(ct)$ are set to true. This case accounts for the fact that if the same parent team is split multiple times by the same `splitTeam`, then the child abstract team created for the `splitTeam` represents multiple concrete teams. (Note that we do not specify how it is to be determined that a `splitTeam` may be executed more than once; it can be done using context-sensitive pointer analysis, detecting when multiple contexts exist for the enclosing method, or using a global control flow analysis.) Otherwise, $multiteam(ct)$ is set to false.
- If `splitTeamSharedMem` is used, then $locality(ct)$ is set to the lower bound of $local$ and $locality(pt)$. Otherwise, $locality(ct)$ is set to $locality(pt)$.
- $level(ct)$ is set to one more than $level(pt)$
- $parent(ct)$ is set to pt
- $children(ct)$ is set to empty
- $multilevel(ct)$ is set to false, unless indicated otherwise by the above rules

Note that the restrictions that an abstract team may not share a creator with any of its ancestors or siblings ensure that the set of abstract teams has finite size and that team analysis eventually terminates.

- A `partition` or `teamsplit` statement takes in a team object as an argument. In the abstract analysis, this object corresponds to a points-to set of abstract locations, each of which has a team set. Let S be the set of all abstract teams in any of these team sets. Then the team set of the `partition` or `teamsplit` body is set to the union of the children of all abstract teams in S, as well as any teams u in S for which $multilevel(t)$ or $multiteam(t)$ is true.

Abstract semantics rules for the remaining Titanium language constructs are relatively straightforward, so we do not discuss them here.

The abstract semantics rules are applied over the entire program until no changes are made to the abstract state. Since the set of abstract teams is finite, as assured by the rules for team allocation expressions and `splitTeam` calls, the team set for each program operation grows monotonically[4], and the abstract semantics rules take finite time to apply, team analysis is guaranteed to terminate eventually.

3.4 Discussion

The pointer portion of team analysis is necessary to determine how abstract teams are passed around by program operations such as assignments. As such, the analysis must track team abstract locations as well as any abstract locations that can transitively reference a team location. Any other information computed by pointer analysis is discarded, so it may be possible to optimize the analysis by ignoring program operations that can be conservatively determined not to have any effect on team analysis.

In addition, team sets are likely to be used by hierarchical pointer and concurrency analysis only to analyze collective operations such as broadcasts and barriers. As a result, team analysis can be further optimized by only maintaining team sets for collective operations and operations relevant to team analysis itself, such as team allocation expressions, team abstract locations, methods, and `partition` and `teamsplit` statements.

While we have defined team analysis in terms of context-insensitive, flow-insensitive pointer analysis, it can be implemented on top of context-sensitive or flow-sensitive pointer analysis as well with little change to the rules above.

[4]This is not quite true, since the rules for allocation include a case where teams are deleted from the analysis as well as any team sets. However, this is just an optimization; we can leave the teams in those team sets and just ignore them in the analysis, in which case team sets would only grow monotonically. Since both cases are equivalent, the fact that the unoptimized case terminates implies that the optimized version does as well.

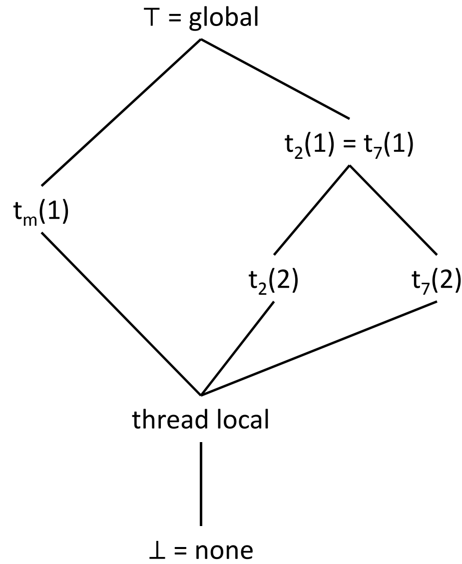

Figure 3: Example of a team lattice.

Note that thread-aware pointer analysis, such as that described in §2.2, is unnecessary; team objects are constructed collectively, and when a team object is used in a **partition** or **teamsplit**, the team library automatically uses the thread-local counterpart of the passed-in team object. Thus, it does not matter on which thread the original object resides.

3.5 The Team Lattice

Once team analysis determines the set of teams in a program and their relationship to each other, a *team lattice* is constructed to represent this information.

The team lattice has a single element for each canonical abstract team in the program; if two abstract teams share the same canonical team, they share a lattice element, even if they differ in other attributes. Edges in the team lattice represent parent-child relationships between abstract teams as computed by the team analysis. A new lattice element is added to represent thread local operations, and an edge is added from that element to each team that has no children. A minimal $\perp$ element represents null operations, and an edge is added from it to thread local. Finally, the maximal element $\top$ corresponds to the global team. Figure 3 shows an example of a team lattice.

Each path from the top of the lattice to thread local represents a distinct team hierarchy. The height of the lattice is the maximum height of all team hierarchies plus one, and we use h_{lat} to denote this value.

3.6 Example

As a concrete example of team analysis and computation of the team lattice, consider the following code that executes in the context of the global team:

```
1 Team a = Ti.defaultTeam();
2 Team b = new Team();
```

```
3 Team c;
4 b.splitTeam(2);
5 teamsplit(b) {
6   b.myChildTeam().splitTeamSharedMem(Ti.thisProc());
7   c = new Team();
8   c.splitTeam(2);
9 }
```

The team analysis determines that a references the global machine team abstract location a_m with the associated abstract team $t_m(0)$, b references an abstract location a_2 with associated team $t_2(0)$, equal to $t_m(0)$, and c references an abstract location a_7 with assoicated team $t_7(1)$, equal to $t_2(1)$. Furthermore, it determines that a_2 has a child abstract location a_4, associated with the abstract team $t_2(1)$, a_4 has a child abstract location a_6 associated with the local abstract team $t_2(2)$, and that a_7 has a child abstract location a_8, associated with the abstract team $t_7(2)$. Finally, team analysis determines that all expressions in lines 1-5 are in the context of team $t_m(0)$ and that those in lines 6-8 are in the context of team $t_2(1)$.

As a result, there are three team hierarchies in this code: the default, machine hierarchy (t_m), the hierarchy created on line 2 and split at lines 4 and 6 (t_2), and the hierarchy created at line 7 and split at line 8 (t_7). The root of each team hierarchy is the global team, and the leaves are individual threads. Including these two levels, the machine team has three levels, with the second level (denoted by $t_m(1)$ using zero-indexing) local, meaning that all threads in a given team at that level share memory. Team t_2 has four levels, with the third level ($t_2(2)$) local. Team t_7 also has four levels and is equal to t_2 at the second level (i.e. $t_2(1) = t_7(1)$). The resulting team lattice is illustrated in Figure 3.

4 Conclusion

In this report, we discussed the information about teams in a recursive single program, multiple data (RSPMD) program that is necessary for client analyses such as pointer and concurrency analysis. We described a basic framework for a team analysis to compute this information. In the future, we hope to define and implement a complete team analysis based on the framework presented here. We also intend to investigate whether a dynamic, runtime analysis can produce more precise results than the static analysis suggested here.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. pages 35–46, 1999.

[3] A. Darte and R. Schreiber. A linear-time algorithm for optimal barrier placement. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 26–35, New York, NY, USA, 2005. ACM.

[4] M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing compiler. Master's thesis, McGill University, Montreal, July 1993.

[5] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in Titanium. In *Supercomputing 2005*, November 2005.

[6] 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.

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

[8] A. Kamil and K. Yelick. Hierarchical additions to the SPMD programming model. Technical Report UCB/EECS-2012-20, University of California, Berkeley, 2012.

[9] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2000.

[10] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *International Static Analysis Symposium*, San Diego, California, June 2003.

[11] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[12] M. O'Boyle. and E. Stöhr. Compile time barrier synchronization minimization. *Parallel and Distributed Systems, IEEE Transactions on*, 13(6):529–543, Jun 2002.

[13] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. *SIGPLAN Not.*, 30(8):144–155, 1995.

[14] 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.