1 Overview of Titanium

Titanium is a language and system for high-performance parallel scientific computing. Titanium uses Java as its base, thereby leveraging the advantages of that language and allowing us to focus attention on parallel computing issues.

- Performance: The user has sufficient control that it should be possible to write high performance code.
- Optimization: Enable compile-time analysis and ooptimization through language constructs.
- Predictability: The programmer should have a clear model of how well the program will be transalted by the compiler. (In sharp contrast are compilers for languages like C, which may perform very sophisticated analysis on pointers to transform loops, but a slight variation on the program may make it unalyzable and therefore unoptimized.)
- Safety: No less safe than Java. When possible, errors are detected at compile-time, rather than runtime.
- Ease of programming: Some higher level constructs are included to simplify certain programming tasks, in particular for many scientific codes.

The main additions to Java are immutable classes, multi-dimensional arrays, an explicitly parallel SPMD model of computation with a global address space, and zone-based memory management. Titanium gives its users access to modern program structure through the use of object-oriented technology, while still providing a transparent performance model of the underlying parallel machine.

2 Sequential Titanium: A Brief Tour of Java

Titanium is, for the most part, an extension of Java 1.1, so most Java programs can be compiled and run as single processor Titanium programs. For example, the following "hello world" program is a correct program with the expected behavior in either language.

```
class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello, world");
    }
}
```

One of the few restriction of Java in Titanium is that program may not create threads. This also means that thread-based Java libraries, such as the Java AWT, cannot be used in Titanium programs.

3 Creating Parallelism

3.1 SPMD Execution Model

Titanium is a Single Program Multiple Data (SPMD) language, meaning that there is a fixed number of processes associated with a running Titanium program. Titanium processes share a common address space with other processes in the same program, but because they cannot be created dynamically (like Java threads, for example) we will refer to them as processes rather than threads. The number of processes is determined by environment-specific information, and is not part of the language.[1]

A simple mental model for an SPMD environment is that a copy of the main function is run on each of the processes. The processes are all running the same code, but unlike a data-parallel language, there is nothing keeping them tightly synchronized on a statement-by-statement basis. Moreover, different processes in the same program may take different branches or execute loops a different number of times. As noted above, most Java programs can be used as sequential Titanium programs, but their behavior on multiple processors may be somewhat surprising. In the "hello world" example above, if the code were compiled and run on multiple processors, each process would separately print "hello" on the screen.

3.1.1 Control in SPMD Programs

Some standard methods are provided as part of the Titanium class library (named `Ti`), for controlling SPMD programs.

- `Ti.numProcs()` is the total number of processes associated with the current program execution.[2]
- `Ti.thisProc()` is the process number of the executing process. Processes are numbered from 0 to `Ti.numProcs()-1`.

The total number of processes is often useful for pairwise communication between processes or for allocating data structures that scale with the number of processes. We will see examples of both of these after describing the use of shared data. A simple example of using the process numbers comes from writing a parallel version of "hello world."

```
class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello from processor " + Ti.thisProc());
    }
}
```

A common idiom in SPMD programs is to have one processor, usually processor 0, perform some activity such as I/O by branching on the processor number:

```
   if (Ti.thisProc() == 0) {
        System.out.println("Hello from one of us.");
    }
```

3.2 Barrier Synchronization

3.2.1 Barriers

Because processes may be at different points in the program text at the same time, it is often useful to synchronize all the processes using barrier synchronization, `Ti.barrier()`. Execution

[1] In the UCB implementation of Titanium, the number of processes is determined by setting an environment variable TI_NODES on shared memory machines and by passing a parameter to the glurun command on the NOW.

[2] *Note to language designers:* The 0.17 version of the language reference uses the team concept. This seems unnecessarily cumbersome without partition.

of a barrier statement causes the executing process to wait until all other processes have reached the same barrier. Many scientific applications are written using a bulk-synchronous model with communication and computation phases separated by barriers:

```
... compute ...
Ti.barrier();
... communicate ...
Ti.barrier();
```

Titanium's notion of barrier is more restricted than in most SPMD languages. In Titanium, all processes must reach the same textual instance of a barrier, while in other languages, a process may proceed through a barrier as long as all processes have reached *some* barrier in the code. For example, the following program is not legal in Titanium:

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

The reason for this restriction is to simplify the compiler analysis of synchronization, which is useful in both earlier detection of synchronization bugs through compile-time checks and better program optimization. The restriction also applies to other global operations, such as broadcasts and reductions, in which there is an implied barrier.

3.3 Introduction to Single Variables

There is support in Titanium for nesting barriers within loops and conditionals. The basic idea is that of a *single* variable or expression, which is guaranteed to have the same value on all processors. For example, a timestep loop in an simulation might be controlled by a single variable that records the current time. If the variable is declared single, the compiler will ensure that all processors have the same value, and barriers may safely be executed within the loop controlled by that variable.

Single will be described in detail in section 5.1. Here, we show the basic syntax for declaring a single variable and note that constants are a useful special case of single expressions. In the follow program fragment, endTime is a single variable, and because 100 is guaranteed to have the same value on all processors, the assignment of endTime to 100 is allowed.

```
int single endTime;
endTime = 100;
```

4 Explicit Communication

Although Titanium is a shared address space language, processes in a Titanium program do not have any shared state when they are created. For programmers used to using thread libraries, the shared state is typically created by passing one or more pointers into the spawned thread; these pointers often point to large data structures that represents all of the shared state between the threads. A less disciplined style is to pass such shared state through globally visible variables. In Titanium, neither of these are options, because the SPMD processes are created implicitly by multiple instance of the main routine, and static variables are not shared, but have independent copies for each SPMD process. Instead, Titanium has small set of operations for explicit communication, including two primitives for creating shared state, `broadcast` and `exchange`, and a library of operations for performing scans and reductions. All of these are global communication operations, requiring participation from all processes. The more frequent point-to-point communication is done implicitly using the shared address space, which is described in Section 5. A typical Titanium application may use explicit communication only to set up shared data structures.

4.1 Broadcast: One-to-All Communication

The basic primitive for creating shared state is to broadcast a value from one processor to all of the others. The value being broadast may be a reference to an object (i.e., a pointer) or a scalar value. The expression:

```
broadcast E from p
```

evaluates expression `p`, which must be an integer in the current range of processors. It then evaluates E on processor p and communicates the resulting value to all other processors, making it the value of the entire expression. To ensure that all processors agree on which one is sending the broadcast, p is required to single; for broadcast expressions, this often happens because p is an integer literal as shown.

```
int count = 0;
if (Ti.thisProc() == 0) {
   count = computeCount();
 }
 count = broadcast count from 0;
```

The value of a broadcast expression is the same on all processors, so it is legal to assign that value to a single variable. This may be useful when obtaining input from the user that will be used to control the problem size or some other global state as in the example below. Note that one must catch any exceptions thrown when reading and converting the input value, because those are only executed by one processor; without the catch clause, some processor could reach the broadcast while others did not.

```
int single count = 0;
int myCount = 0;
if (Ti.thisProc() == 0) {
  try {
     DataInputStream keyboard = new DataInputStream(System.in);
     myCount = Integer.valueOf(keyboard.readLine()).intValue();
   } catch (Exception e) {
     System.err.println("Exception thrown");
   }
 }
 count = broadcast myCount from 0;
```

4.2 Exchange: All-to-All Communication

When creating a shared data structure, one often wants all processors to have a reference to the portion of that data structure stored on each other processor. This could be done by having each processor perform a broadcast, but this is cumbersome and is not the most efficient way to perform such all-to-all communication on some platforms, so Titanium provides a primitive called `exchange`.

Exchange is used to take one value from each of the P processors and fill in a P element array on each processor. Exchange only works on Titanium arrays, which will be described in more detail in a Section 10.

The following is a simple example that uses exchange:

```
int [1d] single allData;
allData = new int [0:Ti.numProcs()-1];
allData.exchange(Ti.thisProc()*2);
```

This code builds an array of `numProcs` integers on each process and fills in the ith element of each array with twice the process number. Thus, when there are four processes, each of the four arrays will contain $[0, 2, 4, 6]$. The four arrays contain identical values after the exchange, but if one process assigns to its own array elements, then nothing is done to automatically keep the four copies consistent. Normally, one does not exchange primitive values, but rather references to objects to create a distributed data structure as will be described in Section 5.

4.3 Basic Scans and Reductions

Broadcast and exchange operations are often used at the beginning of computations to set up data structures. At the end of a program or a phase of computation, one often needs to accumulate results from other processes. These may be done by reading and writing shared variables, but Titanium also provides a convenient set of library routines for reduction and scan operations. A reduction, generally speaking, applies some associative binary operator to an array of elements and produces result of applying that operator to all the elements in the array. For example, a reduction using the add operation would produce a sum of the elements. Titanium does not enforce the requirement that the operator be associative, but leaves the order of evaluation unspecified to allow for tree-based or other computational patterns.

In the simplest form of a reduction operation, all processors receive the final result. For example, in the statement below, all processors will get the sum of 0 through Ti.numProcs()-1 in the variable sum1.

```
int sum1 = Reduce.add(Ti.thisProc());
```

In the next example, the reduction operation takes a second argument, which is the process number of the process that will receive the result. (As in broadcast expressions, this parameter must be single to ensure that all processes agree on which one will receive the result.) In this example, process 1 will receive the sum and all other processors will recieve the default value 0. Zero is the default value for numeric operators and false the default for boolean ones. This form of reduction may be faster than the first form, because the final result need not be broadcast to all processors.

```
int sum2 = Reduce.add(Ti.thisProc(), 1);
```

Scans are similar to reductions, but every processor receives a partial result, depending on their processor number. In the following example, `partsum` on processor 0 is 0, 1 on 1, 3 on 2, and 6 on 3.

```
   int partsum = Scan.add(Ti.thisProc());
```

These examples all show scans and reductions using the `add` operation on integers. Titanium also has operations `mult`, `max`, and `min`, which like `add`, may be applied to `int`, `long`, or `double` types. In addition, there are operations `or`, `xor`, and `and` which an be applied to `int`, `long`, or `boolean`. All of these have the usual meanings. Note that `add` and `mult` on floating point values are not associative, so they may produce different results on different platforms or even on multiple runs on the same platform.

4.4 Generic Scans and Reductions

In addition to these built-in operations, Titanium provides a framework for defining your own operators for scans and reductions. To define your own scan or reduction on a built-in type (int, long, or double), you need to "implement" the IntOp, LongOp, or DoubleOp interface. For example, the DoubleOp interface is:

```
interface IntOp {
   int eval (int x, int y);
}
```

A implementation of this interface, to compute the sum of squares of doubles is:

```
public class SumSquare implements DoubleOp {
  public double eval(double d1, double d2) {
    return d1*d1 + d2*d2;
  }
}
```

This example is purely for demonstration; a simpler way to compute the sum of squares would be to pass the square of each value into the add scan operation.

More interesting generic operations can be defined over arbitrary objects. The program in Figure 1 shows a simple matrix class for 2x2 matrices, one on which multiply and add operations are defined. The two operations are defined in separate classes, because they must both be named `eval` for use in Reductions and Scans. The code here shows how this matrix class code be used with reductions.

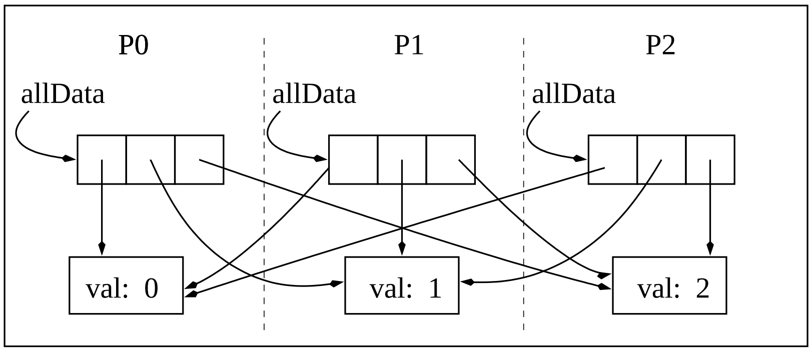
```
// File Reduction.ti (fragment of the main function shown here)

    Mat2x2 ones = new Mat2x2(1,1,1,1);
    Mat2x2 matprod = (Mat2x2) Scan.gen(new Mat2x2Mult(), ones);
    Mat2x2 matsum = (Mat2x2) Scan.gen(new Mat2x2Add(), ones);
```

5 Global Address Space (Implicit Communication)

Explicit communication can be an efficient and elegant way of coordinating a group of processes, but point-to-point communication between individual pairs of processors may be done implicitly through a global address space in Titanium. In comparison to other point-to-point communication systems, such as send/receive messages, the implicit communication model is useful when

```
// File Mat2x2.ti
package mat2x2;
import java.io.*;

public class Mat2x2 {
  double [] mat;

  public Mat2x2 (double a11, double a12, double a21, double a22) {
    mat = new double[4];
    mat[0] = a11;
    mat[1] = a12;
    mat[2] = a21;
    mat[3] = a22;
  }
}

// File Mat2x2Mult.ti
package mat2x2;
public class Mat2x2Mult implements ObjectOp {

  public Object eval (Object m1, Object m2) {
    Mat2x2 result = new Mat2x2(0,0,0,0);
    double a [] = ((Mat2x2) m1).mat;
    double b [] = ((Mat2x2) m2).mat;
    double c [] = result.mat;
    c [0] = a[0] * b[0] + a[1] * b[2];
    c [1] = a[0] * b[1] + a[1] * b[3];
    c [2] = a[2] * b[0] + a[3] * b[2];
    c [3] = a[2] * b[1] + a[3] * b[3];
    return result;
  }
}

// Mat2x2Add.ti
package mat2x2;

public class Mat2x2Add implements ObjectOp {
  public Object eval (Object m1, Object m2) {
    Mat2x2 result = new Mat2x2(0,0,0,0);
    double a [] = ((Mat2x2) m1).mat;
    double b [] = ((Mat2x2) m2).mat;
    double c [] = result.mat;
    c [0] = a[0] + b[0];
    c [1] = a[1] + b[1];
    c [2] = a[2] + b[2];
    c [3] = a[3] + b[3];
    return result;
  }
}
```


Figure 1: Program to define a 2x2 matrix class and associated addition and multiply operations that may be used in reductions.



The references act like traditional Java references; they may be dereferenced and reassigned.

```
if (Ti.thisProc() == 0) {
  ((Boxed) allData[0]).val += 10;
  ((Boxed) allData[1]).val += 20;
}
```

[3] *Note to language designers:* This would change if the semantics of static were modified so that a static variable had only one logial copy; in Titanium, each process has its own copy of static.

The first assignment changes the value inside the box on processor 0, while the second changes the value in the box on processor 1, even though it is executed by processor 0. After these assignment, all processors will still see consistent values for the elements in the data structure, because there is a single copy of each of the boxed integers.

If, however, one processor reassigns one element of the array, that change is reflected only in its own array. For example, both of the assignments below will result in new boxed integers which will be stored in processor 0's array, while the other processor's arrays will be unchanged.

```
if (Ti.thisProc() == 0) {
  allData[0] = new Boxed(4);
  allData[1] = new Boxed(5);
}
```

6.1 Single Variables

The above restriction, that all processes must reach the same textual instance of a barrier, is checked by the compiler to avoid having synchronization errors arise at runtime. This complicates some of the analysis alone, is not enough to allow for robust compile-time analysis of global synchronization constructs. Consider, for example, the following program fragment:

```
while (x < 100) {
   ... compute ...
   Ti.barrier();
 }
 Ti.barrier();
```

Unless the compiler can determine that `x` has the same value on all processes after each iteration of the loop, it cannot prove that the program is correctly synchronized, since some process may exit the loop after fewer iterations than the others and execute the barrier after the loop while the others are still within the loop. The compiler needs information about the variable `x` in this case, namely, that it will have the same value on all processes.

A variable that can be proven to have the same value on all processors is called a *single* variable. In Titanium, global synchronization constructs like barrier may only be called from within a branch or loop if the controlling expressions are single. Literals like `100` in the above example are trivially single, since they have the same value on all processes, but for program variables, they must be declared as single. For example, the above program fragment would be legal only if `x` were declared to be single as in the following declaration:

```
int single x;
```

Once `x` has been declared single, it may only be assigned in blocks of code that are executed by all processes at the same time, i.e., blocks controlled by expressions of single variables. This ensures that `x` will have the same value on all processors whenenever a global synchronization point is reached.

6.1.1 Single Methods

Limiting global synchronization contructs to blocks of code controlled by single variables is not quite sufficient. In the following example, the method `allStep` will run correctly only if it is called by all processes simultaneously.

```
void allStep() {
  ... compute ...
  Ti.barrier();
}
```

A method that must be called by all processes is called a *single* method. Single methods are all those methods that call global synchronization primitives such as barrier, or call other single methods. Titanium allows programmers to document this property by adding a type annotation to the signature, which is either `single` or `sglobal`.

```
single void allStep() {
  ... compute ...
  Ti.barrier();
}
```

Although explicit annotation of single methods is allowed, it is not required. The compiler will infer singleness of a method by analyzing the body of the method. It is important for programmers to understand which methods are single, though, because when building complex distributed data structures, they are likely to see error messages from the compiler regarding single methods.

Single methods may only be executed within other single methods; the `main` method is implicitly single. If a call to a single method appears in a conditional or loop, the controlling expressions must also be single. For example, if `allStep` is single as above, whether declared explicitly or not, then it may be invoked as follows:

```
void allCall() {
  int single x = 0;
  allStep();       // legal, but implies that allCall must be single
  while (x == 0) {
     allStep();    // legal because x is single
  }
}
```

Understanding single methods and variables is important in writing SPMD programs in any language. In Titanium, it is needed to understand errors from the compiler, while in other languages or systems, it is needed to avoid writing programs with synchronization bugs.

6.1.2 Single Syntax

The syntax for designating variables and methods as single can be a bit complicated. The basic rule for variables is that the single keyword goes after the type and before the variable name in a declaration. This is also true when the variable is an argument name in to method signature. See the declaration of `singleVar` in the code below. As described above, single methods are automatically determined by the compiler; the declaration of `doSingle` below shows an optional use of single method declaration. The example also points out that non-single methods may be invoke anywhere, including within a conditional controlled by a single expression. Similarly, single methods may invoke non-single ones, but the reverse is not allowed.

```
public static void main (String [] argv) {  // main is implicitly single
  int single singleVar = 0;
  if (singleVar == 0) {
```

```
    doSingle();        // legal because ‘‘singleVar == 0’’ is a single
    nonSingle();
  }
}

single static void doSingle() { // single here is optional
  int x = 1;
  Ti.barrier();
  nonSingle();
}

static void nonSingle() {
  int x = 3;
}
```

The conditional expression in the `main` method above is obviously single, because the variable `singleVar` is, and the equality test (==) as well as the constant 0 are as well.

6.1.3 Single Return Values

If the conditional expression above contained a method invocation, then the invoked method would have to return a single result for the entire expression to be considered single. The `returnSingle()` method in the code below has such a signature and is therefore legal in a conditional controlling statements containing barriers. Note that a method returning single is not the same as a single (sglobal) method: the former says that expressions involving the return value will be single, while the latter says that the method may only be invoked in code that will be proven to execute simulateneously on all processors.

```
public static single void main (String [] argv) {
  int single singleVar = 0;
  if (singleVar == returnSingle()) {
    Ti.barrier();
  }
}

static int single returnSingle() {
  int single s = 4;
  return s;
}
```

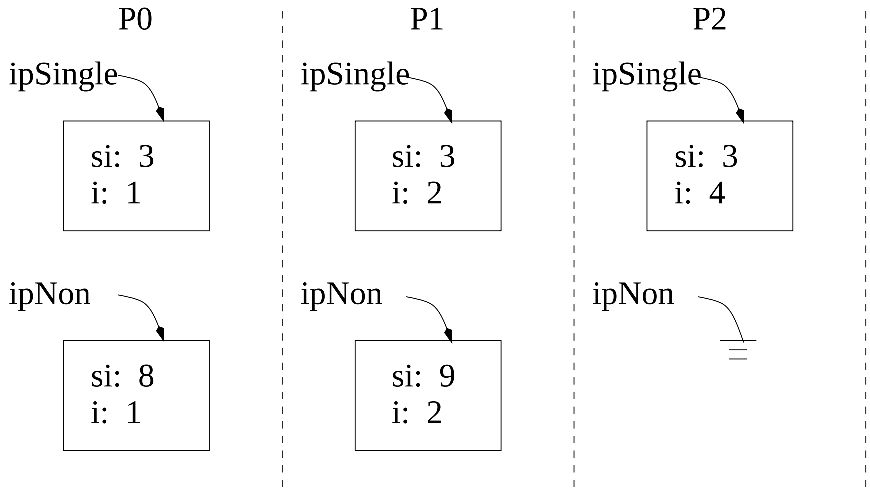
6.1.4 Single and Objects

Although we expect most global control variables to be simple scalar values, it is possible to make objects or fields of objects single. Fields that are either static or automatic may be declared single, for example:

```
class SingleFields {
  int single singleField = 1;
  static int single staticSingleField = 1;
}
```

The single declaration on a field only has an effect when the object it contains is also single. In the example below, because the `ipSingle` variable is single, it must refer to "consistent" objects on in each process, where consistent means that each of the single fields must be identical across processors. Thus, the `si` field of the `ipSingle` objects must have the same value, while the `i` fields are unconstrained. The single declaration on the `si` field has no effect if the object itself is not also declared single. For example, the `ipNon` variable may refer to objects with different `si` (as well as `i` fields) by different processes, or, as shown in the figure below, some processes may have a null reference for `ipNon`, while others refer to a valid object.

```
class IntPair {
  public int single si;
  public int i;
  }

  IntPair single ipSingle;
  IntPair ipNon;
```



This example has objects with only primitive fields. If one of the fields had been a single object, then the contained objects would have to be (recursively) consistent. If a field is an object, but is not single, then as with the integer fields, there are no constraints place on the values of these objects.

When single is used with objects, the user must take care to ensure there are sufficient single declarations to write the desired expressions. For example, to write a method that can be used in a single expression, the return value of that method must be single. The `getSingleValue` in the `BoxInt` class below shows such an example. Since the value returned is a field in the class, that field must be single. To construct an object that contains a single field, one must provide a single value, so in this case the argument to the `BoxInt` constructor must also be single. The intuition is that the program wants to get the same value out of the object on all processors, and must therefore have the same value stored inside.

A programmer is allowed to use a single value as non-single, but not the reverse. Continuing the `BoxInt` example, the method `getNonSingleValue()` is (internally) returning a single value, but the signature does not declare the return value as single. This is legal, but the result of the method invocation will not be viewed as single by the system.

```
class BoxInt {
  private int single val;
  BoxInt(int single x) {
    value = x;
  }

  int single getSingleValue() {
    return val;
  }

  int getNonSingleValue() {
    return value;
  }
}
```

One may use `BoxInt` objects in expressions that control barrier-containing blocks of code. For example:

```
  BoxInt biNon = new BoxInt(3);
  BoxInt single biSingle = new BoxInt(4);
  if (biSingle.getSingleValue() == 3) {
     Ti.barrier();
  }
  if (biSingle == null) {
     Ti.barrier();
  }
```

Note that the non-single `biNon` may not be used in place of `biSingle` in any of the conditional expressions, nor may `getNonSingleValue` be used in place of `getSingleValue`.

6.1.5 Single Program Inputs

As a final note, users may find it useful to declare the input argument to the `main` function as single. The SPMD semantics require that each process gets the same inputs, e.g., the same values for `argv` in the example below:

```
single public static void main (String single [] single argv) {
if (argv == null) {
  Ti.barrier();
} else if (argv[0] == null) {
  Ti.barrier();
}
```

Titanium allows the programmer to declare `argv` as single or not, depending on whether they plan to use the argument variable in single expressions. Both the array and the String elements of the array may be declared single. Making the array single allows for branches on the array reference to be single, as in the first conditional, while making the element single allows branches on the array elements to be single, as in the second conditional.

The above examples are contrived to show the single feature in minimal amounts of code. Some more interesting examples, which will correspond to real program kernels, will be shown in the subsection describing global communication.

7 Synchronization

8 Consistency

9 Static

10 Immutable Classes

11 Titanium Data Structures

11.1 Points and Domains

11.2 Multidimensional Arrays

12 Performance Modeling and Tuning