The Hierarchical SPMD Programming Model

Amir Ashraf Kamil

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-28
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-28.html

April 11, 2011

The Hierarchical SPMD Programming Model

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

July 2, 2009

1 Introduction

In this report, I introduce a new programming model called *hierarchical single program, multiple data* (HSPMD). In this model, each process can divide itself into a team of new SPMD processes, each of which can further divide themselves, resulting in a hierarchy of processes. The division procedure can be parameterized so that the compiler and runtime construct a hierarchy that matches the target machine. Using hierarchical SPMD, a programmer should be able to productively write code that can run efficiently on multiple hierarchical target, including fine-grained architectures such as GPUs and Fleet [7] as well as coarse-grained multicore machines.

2 Background

2.1 Motivation

Numerous hardware architectures have existed in the history of computing, with a variety of structures and feature sets. They have differed in many ways, such as instruction set, cache/memory hierarchy, and even execution model. Despite these differences, sequential programming languages and compilers evolved to abstract software from the underlying hardware, so that the same program can run well on many different architectures. Very few programmers now need to worry about the structure of the machine when writing sequential code.

Parallelism, however, provides new challanges to application portability. Different architectures provide parallelism at different granularities, ranging from fine-grained vector parallelism to coarse-grained SMP machines. In addition, machines tend to be organized hierarchically, with many functional units combined into a processing core, multiple cores combined into a single chip, multiple chips combined in a multi-socket SMP node, and multiple nodes residing in a compute cluster. Each hierarchy level is associated with both computational units and memory.

My goal is to produce a programming model that is almost as productive as sequential programming but still produces programs that run efficiently, and more importantly, scalably on a variety of parallel architectures. In order to be productive, the model must hide most of the messy details of parallel programming. Ideally, it should have sequential semantics, but at the least it should not require a programmer to explicitly synchronize data or schedule computation. On the other hand, the programmer has to provide enough information to allow a compiler to map the program efficiently onto a parallel, hierarchical machine. These two conflicting goals must be carefully balanced, and I do not believe that any existing parallel programming model does so well.

```
01 void task<inner> VectAdd::Inner(in float A[N],
02     in float B[N], out float C[N]) {
03   tunable int T;
04   mappar(unsigned int i = 0 : (N+T-1)/T)
05     VectAdd(A[i*T;T], B[i*T;T], C[i*T;T]);
06 }
07
08 void task<leaf> VectAdd::Leaf(in float A[N],
09     in float B[N], out float C[N]) {
10   for (unsigned int i = 0; i < N; i++)
11     C[i] = A[i] + B[i];
12 }
```

Figure 1: Vector Addition in Sequoia.

2.2 The Sequoia Model

The Sequoia language [3] has the same goal of running efficiently on different hierarchical machines. A Sequoia program consists of a hierarchy of tasks that get mapped to the computational units in a hierarchical machine.

Figure 1 demonstrates a simple vector addition program in Sequoia. Sequoia has two types of tasks: *inner tasks* that decompose computations into subtasks and *leaf tasks* that perform actual computation. As in the `VectAdd` task of Figure 1, a task may have inner and leaf variants. A spawn statement such as that in line 5 of Figure 1 is resolved to the appropriate variant when the task structure is mapped to the target machine hierarchy. Tunable parameters such as `T` in line 3 are also specified during mapping, controlling the number of subtasks spawned in line 4. Thus, both the height and the width of the task tree can be adjusted through task variants and tunable parameters.

Communication in Sequoia is very limited. Only parent and child tasks can communicate, through the use of parameters. Parameters can be read-only (`in`), write-only (`out`), or read/write (`inout`) and are passed by value. Write-only and read/write parameters are not allowed to overlap between multiple children of a parent task; however, the compiler does not attempt to prove that this is the case. While these restrictions eliminate the need for synchronization, they make it very difficult to implement algorithms that require more complex communication or access to global data structures.

The actual mapping of a Sequoia task structure to a machine hierarchy is provided by the programmer in a separate file from the source. It should be possible for the compiler and runtime to produce a reasonable mapping based on the target machine, perhaps with some machine-independent constraints provided by the programmer.

One final limitation in Sequoia is its lack of a rich array, domain, and point language. This can make Sequoia code very explicit and difficult to understand, as well as complicate array partitioning, especially when ghost regions are required.

2.3 SPMD

The single program, multiple data model (SPMD) consists of a fixed set of parallel processes that run the same program. The processes can be executing at different points of the program, though collective operations such as barriers can synchronize the processes at a particular point in the program.

The SPMD model was designed for large-scale parallel machines, so SPMD languages do tend to expose some degree of memory hierarchy to the programmer. For example, UPC [2] has two levels of memory hierarchy, while

```
public abstract class Task {
  // This task's parent task.
  public final Task parent() {...}
  // This task's location in the hierarchy, 0 being the top.
  public final int level() {...}
  // This task's number in its team.
  public final int id() {...}
  // This task's inner variant. By default is the same as leaf variant.
  public void inner() { leaf(); }
  // This task's leaf variant. Must be provided in subclass.
  public abstract void leaf();
}
```

Figure 2: The `Ti.Task` class.

Titanium [4] has three levels [5]. However, SPMD languages do not have a concept of task hierarchy. The set of processes remains the same from program start to finish, and the programmer must explicitly divide them into teams if necessary. Most SPMD languages do not allow collective operations on teams of processes.

3 Language

The demonstration language for HSPMD will be an extension of Titanium, or a subset thereof. In this section, I describe the changes required to Titanium.

3.1 Tasks

The `Ti.Task` class encapsulates an HSPMD task. All tasks must subclass `Ti.Task` and provide at least a leaf variant. For now, all arguments and return values are passed between parent and child task through the task object and are pass-by-reference. Tasks may have tunable parameters by value-parameterizing the task class. A task is identified by an ID number within its team, its level in the task hierarchy, and its parent task.

Program execution starts with a single, anonymous task executing the `main` function. New tasks may be created using a structure similar to Sequoia's `mappar`. The tasks created by such a statement form a team executing the same code. The `mappar` statement does not complete until all subtasks complete, so that parent and child tasks cannot execute simultaneously. Implicit team barriers occur at the beginning and end of task execution.

3.2 Collectives

Since a `mappar` construct creates a team of SPMD tasks, collective operations can be performed on these tasks. Since existing Titanium compiler analyses rely on textual alignment of collectives [6, 5], it would be preferable to retain these semantics. I describe two possible alignment schemes for collectives.

3.2.1 Strict Alignment

The Titanium type system enforces alignment on expressions that *may* execute a collective, rather than only on expressions that actually do perform collectives. Compiler inference determines which expressions may execute collectives,

labeling them as *sglobal*. The following constraints apply to Titanium's alignment scheme.

1. If any task reaches a collective operation, all tasks must reach the same textual instance of the collective.
2. If a collective operation may occur during the course of a method call (i.e. the method is sglobal), then the following must hold:
   - all tasks must invoke the method from the same call site
   - the target function of the invocation must be the same for all tasks
   - if the call occurs inside a loop, all tasks must be in the same iteration of the loop
3. If a collective operation may occur during the course of a conditional, then all tasks must take the same branch.
4. If a collective operation may occur during the course of a loop, then all tasks must execute the same number of iterations.

The *single* type system enforces the above alignment in Titanium [8]. Unfortunately, the type system cannot be extended to teams of tasks [1]. I propose an alternate dynamic enforcement scheme in §4.

3.2.2 Weak Alignment

A weaker alignment scheme would only enforce alignment on expressions that actually perform collective operations at runtime. The following constraints would apply.

1. If any task reaches a collective operation, all tasks must reach the same textual instance of the collective.
2. If a collective operation occurs during the course of a method call, then the following must hold:
   - all tasks must invoke the method from the same call site
   - the target function of the invocation must be the same for all tasks
   - if the call occurs inside a loop, all tasks must be in the same iteration of the loop

Since this definition of alignment is weaker than Titanium's, the current Titanium analyses would need to be modified and would likely be less precise. Regardless, a dynamic enforcement scheme for this definition is also provided in §4.

3.3 Optional Features

The following features can be added to the language if they are found to be useful.

3.3.1 Task Suspension

In some algorithms, it is useful to temporarily move up in the task hierarchy. Iterative algorithms in particular can benefit from this by repeatedly performing an iteration in a team of subtasks and moving into a supertask to do some bookkeeping before passing back down to the subtasks. Here, I describe how this feature can be added.

A new `mappar` variant can be introduced that merely creates tasks without passing execution to them as well as a team object representing the new set of tasks. A call to a run method on this object passes execution to the new tasks. The task team can execute a *suspend* collective that returns execution back to the parent task. A resume call on the team restores execution to the subtasks.

Not only does this scheme allow code in a supertask to be executed without destroying subtasks, it allows phased algorithms to be expressed cleanly. A separate team of subtasks for each phase can be created and run when appropriate. The above semantics ensure that no two subteams can execute simultaneously.

3.3.2 Superset Collectives

It may be necessary in some algorithms to perform collectives above the team level. The semantics of such collectives, however, are unclear, since different teams do not necessarily execute the same code. If task suspension is implemented, such operations can be performed hierarchically instead by executing them at the team level, moving up in the task hierarchy, and repeating.

4 Alignment of Collectives

In this section, I describe a scheme for dynamically ensuring collective alignment that works on teams in the absence of *single*. In summary, this is accomplished by maintaining a running hash representing execution history of a task and comparing these hashes when performing a collective operation. I assume that Titanium's *sglobal inference* determines which methods, conditionals, and loops potentially execute a collective.

When a team of tasks is created, each task creates an empty list that records its execution history. In addition, a hash of this list is maintained, initially set to some value h_0. The following operations update the list, with the hash updated accordingly:

- On a call to an sglobal method, an entry is added to the list with the method's ID and the callsite.
- On an sglobal branch, an entry is added recording the location of the conditional and the branch taken. (Depanding on the alignment semantics, this information may be unnecessary and therefore elided.)
- On an sglobal loop iteration, an entry is added recording the location of the loop and the iteration number. (It may be necessary to use a different representation for loop iterations for performance reasons.)
- On reaching a collective operation, an entry is added to the list with the location of the collective.

When performing a collective, the hashes for all team members are first compared (e.g. by using a comparison tree). If they match, the collective is executed. If any two hashes differ, however, then execution halts, and the corresponding histories are used to generate an appropriate error message. For performance reasons, the execution history list can be eliminated or reduced in size, at the cost of poorer error messages.

Depending on the exact alignment semantics, the execution history and hash may need to be updated at other times. I consider the two alignment schemes described in §3.2.

4.1 Strict Alignment

Under strict alignment, all sglobal method calls, conditionals, and loops must be tracked and checked, regardless of whether or not they actually perform collectives. As such, the execution history and hash must maintain a record of all such operations, even if they have already completed, so that they can be checked at the next collective. The history and hash can be reset at each collective, however, since it is only necessary to check those calls that occured after the previous collective.

4.2 Weak Alignment

Under weak alignment, only method calls and loop iterations that actually perform collectives need be aligned. Thus, sglobal method calls and iterations that do not perform collectives must be discarded from the execution history and hash once they complete. Since it may not be possible to undo the hash, it may be necessary to keep copies of the hash before each sglobal operation, perhaps in the same data structure as the execution history. As in strict alignment, the history and hash can be discarded at each collective operation. In addition, sglobal branches do not need to be tracked.

5 Conclusion

The hierarchical single program, multiple data model enables the hierarchical decomposition of parallel programs while allowing subsets of the hierarchy to execute cooperatively. Though it may pose some implementation challenges, I believe it strikes a good balance between the restrictiveness of the Sequoia and vanilla SPMD models and the flexibility of general dynamic task parallelism. Future work must be done to demonstrate that applications can be written productively in this model, and that a compiler implementation is feasible.

References

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

[2] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

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

[4] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Technical Report UCB/CSD-04-1163-x, University of California, Berkeley, September 2004.

[5] A. Kamil. Analysis of Partitioned Global Address Space Programs. Master's thesis, University of California, Berkeley, December 2006.

[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] I. E. Sutherland. FLEET - A One-Instruction Computer, August 2005. http://research.cs.berkeley.edu/class/fleet/docs/people/ivan.e.sutherland/ies02-FLEET-A.Once.Instruction.Computer.pdf.

[8] 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.

A Semantic Rules for Strict Alignment

A move to dynamic enforcement of collectives necessitates new semantic rules that describe what Titanium programs are valid. In this section, I propose a set of new rules for strict alignment that approximate the restrictions imposed by the static *single* type system.

The Titanium compiler and runtime enforce restrictions on programs to ensure that collective operations are performed in a legal manner. Unless specified otherwise, restrictions are enforced at runtime, resulting in an error when they are violated.

The following properties are important in checking collective alignment:

1. An exception is *universal* if it is explicitly thrown by a qualified **throw** statement:

   ThrowStatement:
   single throw *Expression* ***;***

   Normal **throw** statements are statically taken to throw non-universal exceptions. A **catch** clause in a **try** statement and the **throws** list of a method or constructor declaration indicate that an exception is universal by qualifying the exception type with **single** (we use the terms *universal catch clause* and *universal throws clause* for these cases). A universal catch clause catches only (statically) universal exceptions; a non-universal catch clause will catch both universal and non-universal exceptions.

2. A statement has *global effects* if it or any of its substatements either *may call* a method or constructor that has global effects, is a **broadcast** expression, or is a **single throw** statement.

3. A method or constructor *has global effects* if it is qualified with **single** (which is a new *MethodModifier* that can modify a method or constructor declaration). In addition, a method or constructor has global effects if any statement of its body has global effects.

4. A method call $e_1.m_1(\ldots)$ *may call* a method m_2 if m_2 is m_1, or if m_2 overrides (possibly indirectly) m_1.

A.1 Restrictions on Statements and Expressions with Global Effects

The following restrictions on individual statements and expressions are enforced by the compiler and runtime:

1. If a method call $e_0.v(e_1, \ldots, e_n)$ may call a method m_1 that has global effects then the dynamic dispatch target must be the same in all processes.

2. Instance initializers and initializer expressions for instance (non-static) variables may not have global effects. (For simplicity, this rule is stronger than it needs to be.)

3. In `broadcast` e_1`from` e_2, e_2 must have the same value in all processes.

A.2 Restrictions on Control Flow

The following rules ensure that execution of all statements and expressions with global effects are executed coherently in all processes:

1. If any branch of an **if** statement, a **switch** statement, or a conditional expression (`?`, `&&`, or `||` operator) has statements (or expressions) with global effects, then all processes must execute the same branch.

2. If the body of a **foreach**, **while**, **do/while**, or **for** loop (or in the case of a **for** loop, the test condition) contains statements with global effects, then all processes must execute the same number of iterations.

3. If the main statement of a try has global effects, then non-universal unchecked exceptions may not be specified in its catch clauses. It is an error if one of its non-universal catch clauses catches an unchecked exception at runtime.

4. Associated with every statement or expression that causes a termination t is a set of statements S from the current method or constructor that will be skipped if t occurs. If S contains any statements with global effects, then:

   - If t is a return, then all processes must terminate the method or constructor with t.
   - If t is a checked exception, then it must be universal. It is a compile-time error if this is not the case.

A.3 Restrictions on Methods and Constructors

The following additional restrictions are imposed on methods and constructors:

1. A method may not have global effects if it overrides or implements a method that does not have global effects. The reverse is not true—an overriding method need not have global effects if the method it overrides or implements does.