Brief Announcement: Dynamic Determinacy Race Detection for Task Parallelism with Futures

Rishi Surendran
Rice University
rishi@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

ABSTRACT

Existing dynamic determinacy race detectors for task-parallel programs are limited to programs with strict computation graphs, where a task can only wait for its descendant tasks to complete. In this paper, we present the first known determinacy race detector for non-strict computation graphs with futures. The space and time complexity of our algorithm are similar to those of the classical SP-bags algorithm, when using only structured parallel constructs such as spawn-sync and async-finish. In the presence of point-to-point synchronization using futures, the complexity of the algorithm increases by a factor determined by the number of future operations, which includes future task creation and future get operations. The experimental results show that the slowdown factor observed for our algorithm relative to the sequential version is in the range of $1.00\times$ – $9.92\times$, which is very much in line with slowdowns experienced for fully strict computation graphs.

1. INTRODUCTION

Current dynamic race detection algorithms for task parallelism are limited to parallel constructs in which a task may synchronize with the parent task [5] or an ancestor task [8, 9]. However, current parallel programming models include parallel constructs that support more general synchronization patterns. For example, the OpenMP depends clause allows tasks to wait on previously spawned sibling tasks and the future construct in X10, HJ, C# and C++11 enables a task to wait on any previously created task to which the waiter task has a reference. Algorithms based on vector clocks [1] are impractical for these constructs because either the vector clocks have to be allocated with a size proportional to the maximum number of simultaneously live tasks (which can be unboundedly large) or precision has to be sacrificed by assigning one clock per processor or worker thread, thereby missing potential data races when two tasks execute on the same worker. In this paper, we present the first known sound and precise dynamic determinacy race detector for non-strict computation graphs with futures.

Our work addresses parallel programming models that can support combinations of functional-style futures and imperative-style tasks. Specifically, our race detection algorithm supports async-finish constructs for imperative-style parallelism and future construct for functional-style parallelism. The statement "`async { S }`" causes the parent task to create a new child task to execute `S` asynchronously with the remainder of the parent task. The statement "`finish { S }`" causes the parent task to execute `S` and then wait for the completion of all asynchronous tasks created within `S`. Each dynamic instance T_A of an async task has a unique *Immediately Enclosing Finish (IEF)* instance F of a finish statement during program execution, where F is the innermost finish containing T_A. A future [6] refers to an object that acts as a proxy for a result that may initially be unknown because the computation of its value may still be in progress as a parallel task. The statement, "`future<T> f = async<T> Expr;`" creates a new child task to evaluate `Expr` asynchronously, where `T` is the type of the expression `Expr`. In this case, `f` contains a handle to the return value (future object) for the newly created task and the operation `f.get()` can be performed to obtain the result of the future task. If the future task has not completed as yet, the task performing the `f.get()` operation blocks until the result of `Expr` becomes available.

The remainder of this brief announcement is organized as follows. Section 2 defines a determinacy race for our programming model. Section 3 presents our approach for determinacy race detection for parallel programs with futures, and Section 4 discusses the experimental results for our race detection algorithm.

2. DATA RACES AND DETERMINISM

We define a data race for programs containing async, finish, and future constructs in terms of the ***computation graph*** [2], and as a preamble to defining determinacy races. A computation graph models the execution of a parallel program as a graph, where each node in a computation graph corresponds to a step, where a step is defined as follows:

Definition 1. *A step is a maximal sequence of statement instances such that no statement instance in the sequence includes the start or end of an async, finish or a get operation.*

The edges in computation graph represent parallel control dependences. There are three different types of edges in a

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

ACM ISBN 978-1-4503-4210-0/16/07.

DOI: http://dx.doi.org/10.1145/2935764.2935815

computation graph (akin to those in a fully strict computation graph):

1. **Continue Edges** capture the sequencing of steps within a task. All steps in a task are connected by continue edges.
2. **Spawn Edges** represent the parent-child relationship among tasks. When task A creates task B, a spawn edge is inserted from the step that ends with the async in task A to the step that starts task B.
3. **Join Edges** represent the synchronization among tasks. When task A performs a get on future B, a join edge is inserted from the last step of B to the step in task A that immediately follows the `get()` operation. Join edges are also inserted from the last step of every task to the step in the ancestor task immediately following the Immediately Enclosing Finish (IEF). A join edge from task B to task A is referred to as ***tree join*** if A is an ancestor of B; otherwise, it is referred to as a ***non-tree join***.

Definition 2. *A step u is said to precede step v, denoted as $u \prec v$, if there exists a path from u to v in the computation graph.*

We use the notation $u \not\prec v$ to denote that there is no path from step u to step v in the computation graph.

Definition 3. *A data race may occur between steps u and v, iff both u and v access a common memory location, at least one of which is a write and $u \not\prec v$ and $v \not\prec u$.*

We say that a parallel program is *functionally deterministic* if it always computes the same answer when given the same inputs. By default, any sequential computation is expected to be deterministic with respect to its inputs; if the computation interacts with the environment (e.g., a GUI event such as a mouse click, or a system call like System.nanoTime()) then the values returned by the environment are also considered to be inputs to the computation. Further, we refer to a program as *structurally deterministic* if it always computes the same computation graph, when given the same inputs. Finally, following past work [7]we say that a program is *determinate* if it is both functionally and structurally deterministic. If a parallel program is written using only async, finish, and future constructs, *and is guaranteed to never exhibit a data race*, then it must be determinate, i.e., both functionally and structurally deterministic. Note that all data-race-free programs written using async, finish and future constructs are guaranteed to be determinate, but it does not imply that all racy programs are non-determinate.

3. RACE DETECTION APPROACH

Our race detection algorithm detects races on-the-fly during a depth-first serial execution of the input program. A dynamic race detector needs to provide mechanisms that answers two questions: for any pair of memory accesses, at least one of which is a write 1) can the two accesses logically execute in parallel? 2) do they access the same memory location? To answer the first question, we introduce a program representation referred to as *dynamic task reachability graph* which is presented in Section 3.1. Similar to most race detectors, we use a shadow memory mechanism to answer the second question which is presented in Section 3.2. Section 3.3 presents a high-level view of our algorithm.

3.1 Dynamic Task Reachability Graph

The dynamic task reachability graph represents the reachability information at task-level instead of step-level. It uses the following three ideas for encoding reachability information between steps in the computation graph of the input program:

- *Disjoint set representation of tree joins:* The reachability information between tasks which are connected by tree join edges is captured using a disjoint set data structure. Two tasks A and B are in the same set if and only if B is a descendant of A and there is a path in the computation graph from B to A which includes only tree-join edges and continue edges. Any m operations on n sets take a total of $O(m\ \alpha(m,n))$ time where α is the inverse Ackermann's function.
- *Interval encoding of spawn tree:* To efficiently store and query reachability information from a task to its descendants, we use a labeling scheme [4], where each task is assigned a label according to the preorder and postorder numbering scheme. The values are assigned according to the order in which the tasks are visited during a depth-first-traversal of the *spawn tree*, where the nodes in the spawn tree correspond to each of the tasks and edges represent the parent-child spawn relationship. Ancestor-descendant relationship queries between task pairs can be answered by checking if the interval of one task subsumes the interval of the other task.
- *Immediate predecessors+significant ancestor representation of non-tree joins:* The non-tree joins in the computation graph are represented in the dynamic task reachability graph as follows:
  - *immediate predecessors*: For each non-tree join from task A to task B, B stores A in the set of predecessors.
  - *lowest significant ancestor*: We define the *significant ancestors* of task A as the set of ancestors of A which has performed at least one non-tree join operation. For each task, we store only the lowest significant ancestor.

3.2 Shadow Memory

Our algorithm maintains a shadow memory M_s for every shared memory location M. M_s contains the following fields

- *writer*, a reference to a task that wrote to M. $M_s.writer$ is initialized to *null* and is updated at every write to M. It refers to the task that last wrote to M.
- *readers*, a set of references to tasks that read M. $M_s.readers$ is initialized to $\emptyset$ and is updated at reads of M. It contains references to all future tasks that read M in parallel since the last write to M. It also contains a reference to one non-future (async) task which read M since the last write to M.

3.3 Algorithm

As the input program executes in serial, depth-first order the race detection algorithm performs additional operations whenever one of the following actions occurs: task creation, task return, `get()` operation, shared memory read and shared memory write. Determinacy races are detected when a read or write to a shared memory location occurs. When a write to a memory location M is performed by step

Benchmark	Description	Input Size	Seq (millisecs)	Racedet (millisecs)	Slowdown
Series-af	Fourier coefficient analysis using async-finish	Size C	483,224	484,746	1.00
Series-future	Fourier coefficient analysis using futures	Size C	487,134	487,985	1.00
Crypt-af	IDEA encryption algorithm using async-finish	Size C	15,375	119,504	7.77
Crypt-future	IDEA encryption algorithm using futures	Size C	15,517	128,234	8.26
Jacobi	2 dimensional 5-point stencil using futures	2048×2048	3,402	27,388	8.05
Smith-Waterman	Sequence alignment using futures	10000	3,488	34,558	9.92
Strassen	Strassen's algorithm using futures	1024×1024	30,811	33,618	5.35

Table 1: Runtime overhead for determinacy race detection. **Seq** is the sequential execution time in milliseconds and **Racedet** is the execution time with race detection enabled in milliseconds. **Slowdown** is the slowdown due to race detection **(Racedet/Seq)**.

u, the algorithm checks if the previous *writer* or the previous *readers* in the shadow memory space may execute in parallel with the currently executing step and reports a race. It updates the *writer* shadow space of M with the current task and removes any task S if $S \prec u$. When a read to a memory location M is performed by step u, the algorithm checks if the previous *writer* in the shadow memory space may execute in parallel with the currently executing step and reports a race. It adds the current task to the set of readers of M and removes any task S if $S \prec u$. Given tasks A and B, our algorithm determines if they can execute in parallel by inspecting the dynamic task reachability graph. The algorithm does this by traversing only the non-tree edges and using the disjoint set and interval labels to answer reachability queries along paths with spawn and tree join edges.

Our race detection algorithm can be implemented to check a program that executes in time T on one processor for determinacy races in $O(T\ (f+1)\ (n+1)\ \alpha(T, a+f))$ time using $O(a+f+n+v\ (f+1))$ space, where a is the number of async tasks and f is the number of future tasks created by the program, n is the number of non-tree joins performed by the program and v is the number of shared memory locations referenced by the program.

4. EXPERIMENTAL RESULTS

The race detector was implemented as a new Java library for detecting determinacy races in HJ [3] programs containing async, finish, and future constructs. The benchmarks written in HJ were instrumented for race detection during a bytecode-level transformation pass implemented on HJ's Parallel Intermediate Representation (PIR). The instrumentation pass adds the necessary calls to our race detection library at async, finish, and future boundaries, future get operations, and also on reads and writes to shared memory locations.

Our experiments were conducted on a 16-core Intel Ivybridge 2.6 GHz system with 48 GB memory, running Red Hat Enterprise Linux Server release 7.1, and Sun Hotspot JDK 1.7. To reduce the impact of JIT compilation, garbage collection, and other JVM services, we report the mean execution time of 10 runs repeated in the same JVM instance for each data point.

The results of our evaluation are given in Table 1. The first column lists the benchmark name. The fourth column (Seq) reports the average execution time of the sequential (serial elision) version of the benchmark, and the following column (Racedet) reports the average execution time of a 1-processor execution of the parallel benchmark using the determinacy race detection algorithm introduced in this paper. Finally, the Slowdown column reports the ratio of the Racedet and Seq values. The slowdowns for Series-af and Crypt-af are comparable to the slowdowns reported for the ESP-Bags algorithm that only supported async and finish, thereby showing that our determinacy race detector does not incur additional overhead for async/finish constructs relative to state-of-the-art implementations. The slowdown for Crypt-future is higher than that of Crypt-af because of two reasons: 1) the additional number of memory accesses due to the future references and 2) the average number of readers stored in the shadow memory is higher, because of the presence of future tasks. The slowdowns for Jacobi, Smith-Waterman, and Strassen (8.05×, 9.92×, and 5.35×) are positively correlated with the number of shared memory accesses, the average number of readers stored in the shadow memory, and 1/Seq.

5. REFERENCES

[1] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *PADTAD '06*, pages 69–78, New York, NY, USA, 2006. ACM.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[3] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *PPPJ '11*, pages 51–61, New York, NY, USA, 2011. ACM.

[4] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC '87*, pages 365–372, New York, NY, USA, 1987. ACM.

[5] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA '97*, pages 1–11, New York, NY, USA, 1997. ACM.

[6] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[7] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, May 1969.

[8] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. *Formal Methods in System Design*, 41(3):321–347, December 2012.

[9] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *PLDI '12*, pages 531–542, New York, NY, USA, 2012. ACM.