Efficient Data Race Detection for Distributed Memory Parallel Programs

Chang-Seo Park   Koushik Sen
University of California, Berkeley
{*parkcs,ksen*}*@cs.berkeley.edu*

Paul Hargrove   Costin Iancu
Lawrence Berkeley National Laboratory
{*phhargrove,cciancu*}*@lbl.gov*

ABSTRACT

In this paper we present a precise data race detection technique for distributed memory parallel programs. Our technique, which we call Active Testing, builds on our previous work on race detection for shared memory Java and C programs and it handles programs written using shared memory approaches as well as bulk communication. Active testing works in two phases: in the first phase, it performs an imprecise dynamic analysis of an execution of the program and finds potential data races that could happen if the program is executed with a different thread schedule. In the second phase, active testing re-executes the program by actively controlling the thread schedule so that the data races reported in the first phase can be confirmed. A key highlight of our technique is that it can scalably handle distributed programs with bulk communication and single- and split-phase barriers. Another key feature of our technique is that it is precise—a data race confirmed by active testing is an actual data race present in the program; however, being a testing approach, our technique can miss actual data races. We implement the framework for the UPC programming language and demonstrate scalability up to a thousand cores for programs with both fine-grained and bulk (MPI style) communication. The tool confirms previously known bugs and uncovers several unknown ones. Our extensions capture constructs proposed in several modern programming languages for High Performance Computing, most notably non-blocking barriers and collectives.

1. INTRODUCTION

In order to achieve scalability and efficient utilization on future Peta and Exascale systems, high performance scientific computing uses hybrid parallelism: the SPMD paradigm used for inter-node coordination is augmented with shared memory approaches for intra-node programming. Many studies showcase the advantages of MPI+X (X=OpenMP, Unified Parallel C [15]), while others use Partitioned Global Address Space (PGAS) languages: UPC [9], Co-Array Fortran, Chapel and X10. In these approaches asynchrony is employed at the shared memory node level as well as for inter-node coordination, resulting in a program where concurrency bugs are likely to occur and are hard to find.

Currently, there are not many tools to help find concurrency bugs in distributed memory scientific programs. Most of the existing work for SPMD programs uses static analysis, e.g. barrier matching [3, 70] or single value analysis [30]. Static analysis requires extensive compiler support, often lacks whole program information and reports a large number of false positives. Debugger based approaches also face challenges finding concurrency bugs due to their non-determinism.

In this paper, we consider data races in distributed memory parallel programs. A parallel program has a data race if there is a reachable state of the program where two threads are trying to access the same memory location and at least one of them is a write. Data races are often considered to be bugs because programs with data races can compute unintended non-deterministic results.

We propose an *active testing* framework, called (*UPC-Thrille*), to find concurrency bugs in distributed memory programs with precision and scalability. Active testing works in two phases. In the first phase, called the *race prediction phase*, active testing performs an imprecise dynamic analysis of a test execution of a program to find a set of pairs of program statements that could potentially race in an (potentially different) execution of the program. In the second phase, called the *race confirmation phase*, active testing tries to confirm the potential data races found in the first phase. Specifically, for each pair of statements reported in the first phase, active testing re-executes the program under a controlled thread schedule in an attempt to bring the program in a state where two threads have reached the pair of statements, respectively, and are about to access the same memory location and at least one of the accesses is a write. If it succeeds, active testing has created an actual program execution exhibiting the data race. Note that active testing gives no false warnings because for each reported data race, it creates a real execution exhibiting the data race; however, it may miss real data races present in the program because a dynamic analysis cannot find data races in parts of the program that has not been executed. We have implemented the framework for the UPC language, which provides several constructs present in modern parallel programming languages: split-phase barriers, non-blocking communication and a memory consistency model which affects message ordering. We present the first implementation of data race detection dynamic analyses able to handle distributed memory and demonstrate scalability up to a thousand cores.

SC11, November 12-18, 2011, Seattle, Washington, USA
Copyright 2011 ACM 978-1-4503-0771-0/11/11 ...$10.00.

In our implementation, we use a lockset-based [54] dynamic analysis in the first phase to identify potential data races. We augment the UPC runtime calls for communication and synchronization with analysis specific instrumentation: this code is generic and can be retrofitted easily into other runtimes. Lockset based analysis requires a central thread to collect and analyze the memory accesses and synchronization operations of all threads. Such an analysis thread is easy to implement for non-distributed shared memory programs. However, for distributed memory parallel programs, a central analysis thread incurs a huge communication overhead; our initial experiments showed that such an implementation of a central analysis thread fails to scale beyond a few nodes. In this paper, we propose several novel techniques to avoid such a central analysis and heavy communication overhead. The experimental results show that our implementation can scale to a thousand of cores.

In our tool, for each task we build a database of "local" events of interest, e.g. memory accesses for the data race analysis. During execution, other remote tasks perform queries on this database in order to detect the actions that can lead to a data race. In Section 5 we discuss scalability optimizations for this phase: efficient search data structures and coaleascing of remote queries in synchronization operations.

The contributions of our work are summarized as follows:

- We propose a distributed dynamic analysis technique to predict and confirm data races in distributed memory parallel programs. Our novel distributed analysis eliminates the need for a central analysis thread; therefore, the technique scales to a large number of nodes.
- We extend the lockset-based data race detection algorithm [54] to handle split-phase barrier and non-blocking collective operations.
- We present and evaluate the first implementation of data race detection on distributed memory.

2. ACTIVE TESTING OF PARALLEL PROGRAMS

Our work in this paper builds on active testing that shows promise both in terms of precise analysis and scalability for distributed memory programs. The active testing methodology has been first introduced by Sen et al. [56, 33] for shared memory programs and it handles at least five classes of common concurrency bugs, namely data races [56], atomicity violations [49], deadlocks [34], missed notifications, and bugs due to relaxed memory models [8]; it can easily be extended to other classes of concurrency bugs. Active testing requires a separate, but similar, implementation for each class of bugs. A key novelty of the approach is that it requires mostly runtime instrumentation (i.e. little or no compiler support) and it can conveniently describe the analysis for each class of bugs using two phases:

1. A predictive analysis phase, where active testing analyzes the execution of a program to generate a set (or database) of tuples—the format of a tuple being specific to the class of the bug. At the end of the execution, active testing makes queries on the set (or database) of tuples such that each result of the query represents a *potential* bug. Again the form of a query is specific to a class of bugs.
2. A testing phase, where active testing re-executes the program under controlled thread schedules in an attempt to check the feasibility of each bug predicted in the predictive analysis phase.

In our previous work we have implemented Active Testing for Java [33] and C/pthreads [32] shared memory programs and applied it to a large number of benchmarks, some containing over 500K lines of code. The tool has been able to find previously unknown bugs in several real-world Java and C/pthreads programs.

We strongly believe that when porting applications from a bulk synchronous MPI style to hybrid implementations that are highly asynchronous, concurrency bugs will become prevalent and efficient tools to assist developers in finding these bugs are required. In this paper we present the design and implementation of the active testing framework for the UPC programming language. As discussed later, UPC contains most of the features present or proposed in modern parallel programming languages and our implementation provides a proof of concept for the generality of the active testing approach.

3. UNIFIED PARALLEL C

Unified Parallel C (UPC) is an extension to ISO C 99 that provides a Partitioned Global Address Space (PGAS) abstraction using Single Program Multiple Data (SPMD) parallelism. The memory is partitioned in a task (unit of execution in UPC) local heap and a global heap. All tasks can access memory residing in the global heap, while access to the local heap is allowed only for the owner. The global heap is logically partitioned between tasks and each task is said to have local affinity with its sub-partition. Global memory can be accessed either using pointer dereferences (load and store) or using bulk communication primitives (`memget(), memput()`). The language provides synchronization primitives, namely locks, barriers and split phase barriers. Most of the existing UPC implementations also provide non-blocking communication primitives, e.g. `upc_memget_nb()`. The language also provides a memory consistency model which imposes constraints on message ordering.

Locks and barriers are common synchronization constructs in shared memory programming models (e.g. pthreads) and previous work [56, 33] has shown how to build concurrency analyses for these primitives. Bulk communication operations and split-phase barriers pose additional challeges to these techniques. In Sections 4 and 5.1 we present new formalisms to accommodate these language primitives into active testing. The additional UPC language constructs have similar counterparts in other languages: split-phase barriers are similar to non-blocking barriers and collectives, non-blocking communication is prevalent across languages. Thus, programs written in UPC exhibit most of the characteristics of other programming models: MPI+OpenMP, X10, and Chapel.

In addition to the built-in synchronization operations, several applications written in UPC implement ad-hoc primitives using strict memory references. Since they impose only constraints on message ordering within a single task, strict memory references per se do not require special handling within the active testing framework. However, their presence usually indicates application specific synchronization mechanisms. In these programs, phase one of active test-

```
29   int build_table (int nitems, int cap,
30       shared int *T, shared int *w, shared int *v)
31   {
32       int wj, vj;
33       wj = w[0];
34       vj = v[0];
35       upc_forall(int i = 0; i < wj; i++; &T[i])
36           T[i] = 0;
37       upc_forall(int i = wj; i <= cap; i++; &T[i])
38           T[i] = vj;
39       upc_barrier;

     }
108  int main( int argc, char** argv )
     {


140      upc_forall(i = 0; i < nitems; i++; i)
141      {
142          weight[i] = 1 + (lrand48()%max_weight);
143          value[i]  = 1 + (lrand48()%max_value);
144      }
145      best_value = build_table(nitems, capacity,
146              total, weight, value );
```

Figure 1: Parallel knapsack implementation with data race.

ing may report false positives, i.e. races that will never be confirmed by phase two. Eliminating these false positives in phase one requires application semantic knowledge and is beyond the scope of any "language only" testing methodology. Furthermore, phase two of active testing will not be able to confirm the false positives and therefore will not report them to users.

Achieving good performance and scalability is a challenge that needs to be addressed when implementing active testing in distributed memory environments: the implementation of the tuple generation and the query phases can be easily impaired by the high network latency. Another contribution of our work, discussed throughout Section 5, is a suite of optimizations for cluster based systems.

4. RACE DIRECTED ACTIVE TESTING FOR UPC

Data races happen when in an execution two tasks are about to access the same memory location, at least one access is a `write`, and no ordering is imposed between these concurrent accesses. Figure 1 is a partial listing for a UPC program that computes the "0-1 knapsack" problem in parallel using dynamic programming. Although not apparent at first look, there are two data races in this program that can lead to incorrect results. The first data race is between lines 33 and 142, and the second between lines 34 and 143. Since a `upc_forall` statement is not a collective operation, there is no ordering enforced between the write to `weight[0]` in line 142 and the read from it in line 33. If the read happens before the write, the table may be incorrectly initialized and result in a wrong computation. The second race is similar to the first. Ironically, the program has been assigned for years as homework for graduate parallel programming courses at UC Berkeley. The bug has been independently reported by students at the time of this writing and detected by our tools.

Our methodology for active testing for UPC proceeds in two phases. We next describe these two phases in detail.

4.1 Phase I: Race Prediction Phase

In phase one of active testing, we run an imprecise dynamic analysis on an execution of the program to find potential data races that are present in the program. The analysis is a variant of a lockset based algorithm [54, 10, 47, 63]. The analysis checks if two tasks could potentially access a memory location without holding a common lock. Specifically, the analysis observes all memory accesses that happen during an execution of the program and records the locks held during each such access. If there exists two accesses to the same memory location by different tasks and if a common lock is not held during the accesses and if at least one of the accesses is a write, then the analysis reports a potential data race. The analysis also reports the pairs of statements where the tasks access the memory location, respectively. Formally, the set of potential data races pairs reported by the analysis is defined as follows:

Definition 1 ($\mathcal{D}_{P,E}$: Set of Potential Data Race Pairs). *Given an execution E of a program P, let us denote a memory access* event *by a task in the execution by $e = (m, t, l, a, s)$, where*

1. *m is the memory address range that is being accessed,*
2. *t is the task accessing the memory address range.*
3. *l is the set of locks held by t at the time of access,*
4. *$a \in \{READ, WRITE\}$ is the access type, and*
5. *s is the label of the program statement that generates the memory access event.*

Let $\mathcal{A}_{P,E}$ be the set of all memory access events in the execution E. Then the set of potential data race pairs reported by the analysis is

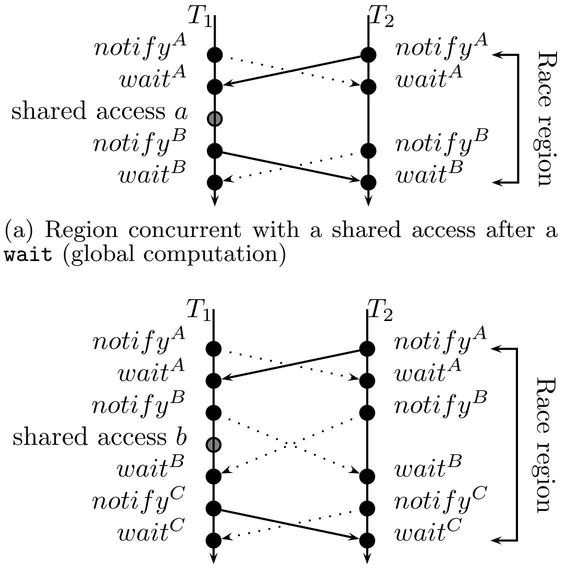
$$\begin{aligned}\mathcal{D}_{P,E} = \{(s_1, s_2) \mid \exists e_1, e_2 \in \mathcal{A}_{P,E} \text{ such that } \\ e_1 = (m_1, t_1, l_1, a_1, s_1) \ \wedge \ e_2 = (m_2, t_2, l_2, a_2, s_2) \\ \wedge \ m_1 \cap m_2 \neq \emptyset \ \wedge t_1 \neq t_2 \ \wedge \ l_1 \cap l_2 = \emptyset \\ \wedge \ (a_1 = WRITE \vee a_2 = WRITE)\} \ .\end{aligned}$$

Note that a race pair reported by the analysis can be a false warning because the analysis does not check if the two accesses are ordered by a synchronization operation. The analysis simply checks if the program adheres to the idiom that every memory access is consistently protected by a lock. As such, the analysis can report data races that did not actually happen in the execution E, but could happen if the program is executed under a different thread schedule. This predictive power of the analysis is crucial for increasing the coverage of our active testing technique.

In lieu of lock based synchronization, scientific programs tend to use barrier synchronization: a barrier partitions the program execution into different phases and prevents a task from advancing to the next phase before all other tasks have completed the phase. Due to this kind of synchronization, our phase one analysis reports a large number of false warnings. In order to eliminate these false warnings, we propose a modification to our analysis, called *Barrier Aware May-Happen-in-Parallel Analysis.* We next describe this modified analysis for phase one.

Barrier Aware May-Happen-in-Parallel Analysis

In order to hide communication latency in clusters, split phase barriers are provided in the UPC language. Nonblocking collectives serve a similar purpose in other languages. A split phase barrier in UPC is implemented by



(a) Region concurrent with a shared access after a `wait` (global computation)

(b) Region concurrent with a shared access after a `notify` (local computation)

Figure 2: The regions of other tasks concurrent with a shared access

a pair of calls `upc_notify` and `upc_wait`. As long as there are no data conflicts, tasks can execute arbitrary code in between this pair of calls. In UPC, a task cannot progress from a wait call until notifications from all other tasks arrive. To illustrate which instructions can execute concurrently, we formally define the *happens-before* [36] relation.

Definition 2 (Happens-before: →). *The happens-before relation → on a set of events is the smallest relation satisfying the following conditions.*

1. *$e_1 \to e_2$ if e_1 and e_2 are events of the same task and e_1 precedes e_2 in the execution,*
2. *$e_1 \to e_2$ if e_1 is a message send event (*`notify`*) and e_2 is the receive event of the same message (matching* `wait`*), and*
3. *$e_1 \to e_2$ if there exists an event e such that $e_1 \to e$ and $e \to e_2$.*

Events e_1 and e_2 are concurrent if they are not related by the happens-before relation, i.e. if $e_1 \not\to e_2$ and $e_1 \not\to e_2$.

For our purposes, we only consider split-phase barriers, since a single-phase barrier can be expressed as a consecutive `notify` and `wait` with no statements in between.[1] According to the happens-before relation, a shared access a can happen concurrently with an access from another task in the region starting from the notify that matches the first wait before a and ending at the wait matching the first notify after a. Figure 2 illustrates this scenario. The arrows denote the happens-before relation induced by the barrier synchronization behavior, with the events affecting the ordering of the shared access indicated as solid arrows. The upper diagram shows the region of task T_2 that can happen in parallel with a shared access of task T_1 between a `wait` and `notify`. Following the split-phase barrier semantics, the shared access cannot happen before all other tasks have notified $wait^A$ of T_1, and similarly other tasks cannot go beyond $wait^B$ before T_1 executes $notify^B$. The lower diagram shows that a shared access of T_1 between a `notify` and `wait` can happen in parallel with a larger region of other tasks.

Based on the happens-before relation of `notify` and `wait` statements, we derive a *may-happen-in-parallel* relationship between program blocks and incorporate barrier awareness in the race detection analysis.

Definition 3 (May-happen-in-parallel: ||). *Let each task t_i have a program phase counter $p_i \in \mathbb{N}$. Initially, $\forall i.\ p_i = 0$. After a task executes each* `notify` *and* `wait`*, the phase counter is increased by 1. Two phases p_1 and p_2 may happen in parallel, denoted as $p_1 \mid\mid p_2$, if*

$$p_2 \in \left[2\left\lfloor\frac{p_1}{2}\right\rfloor - 1, 2\left\lfloor\frac{p_1+1}{2}\right\rfloor + 1\right] .$$

The formula in Definition 3 unifies the fact that even phases (global computation) can race with phases ±1 and odd phases (local computation) can race with phases ±2. Note that $p_1 \mid\mid p_2$ is a necessary condition for all statements in p_1 to be concurrent with statements in p_2. We now extend the lockset algorithm with the barrier aware may-happen-in-parallel analysis.

Definition 4. (Set of Barrier Aware Potential Data Race Pairs: $\bar{\mathcal{D}}_{P,E}$) *We extend the memory access event in Definition 1 by adding a field for the program phase p of task t: $e = (m,t,l,a,p,s)$ The set of potential data race pairs reported by our analysis is*

$$\begin{aligned}\bar{\mathcal{D}}_{P,E} = \{(s_1,s_2) \mid \exists e_1,e_2 \in \mathcal{A}_{P,E} \text{ such that } p_1||p_2 \wedge \\ m_1 \cap m_2 \neq \emptyset \wedge t_1 \neq t_2 \wedge l_1 \cap l_2 = \emptyset \wedge \\ (a_1 = WRITE \vee a_2 = WRITE)\} .\end{aligned}$$

The first phase of active testing executes the program once to build the set of possible data races $\bar{\mathcal{D}}_{P,E}$ from the above definition. Each task builds a trace of memory accesses $e = (m,t,l,a,p,s)$ with respect to the barrier phases and set of locks held at that program point. Each task maintains this trace for the portion of the global heap with local affinity. For any global memory reference in T_1, phase I performs a query operation on the trace of the task responsible for maintaining state for that region, e.g. T_2. If an outstanding conflicting access is found on T_2, the detection algorithm has identified a potential data race, i.e. it has found a statement pair (s_1,s_2) in $\bar{\mathcal{D}}_{P,E}$.

4.2 Phase II: Race Confirmation Phase

From phase I, we get the set $\bar{\mathcal{D}}_{P,E}$. For each statement pair in $\bar{\mathcal{D}}_{P,E}$, there is a possibility that it may be a false positive due to lack of application specific semantic information, e.g. usage of custom synchronization operations. In the second phase of active testing, we re-execute the program and actively control the thread schedule in an effort to confirm the real data races. Specifically, for each statement pair (s_1,s_2) in $\bar{\mathcal{D}}_{P,E}$, we try to create an execution state where two tasks are about to execute s_1 and s_2, respectively, and they are about to access the same memory location, and at least one of the accesses is a write. Such an execution is evidence that the data race over the statements s_1 and s_2 is a real race. The second phase works as follows. For each

[1] This is how the Berkeley UPC Runtime defines single-phase barriers: `#define upc_barrier (x) { upc_notify(x); upc_wait(x); }`

statement pair (s_1, s_2) in $\bar{\mathcal{D}}_{P,E}$, we execute the program. During the execution, we pause the execution of any task that reaches s_1 (or s_2). If some task, say t_2, reaches s_2 then we check if there exists another task t_1 such that t_1 is currently paused at statement s_1 and t_1 is trying to access an overlapping memory region with t_2. If such a task t_1 exists, then active testing reports an actual data race. Otherwise, we pause task t_2 at statement s_2 (or s_1 respectively). This process is repeated until program termination. Note that the above execution could end up in a deadlock situation if we end up pausing all the tasks in the program. In order to avoid such deadlocks, we resume each paused task after some time T. If T is large, it can significantly slow down the execution. Conversely if T is small, we may miss a real race because we did not give enough time for the other tasks to catch up and create a data race. In our experiments, we found that $T = 10ms$ is a suitable pause time, which does not slowdown an execution too much while confirming all the real data races.

5. IMPLEMENTATION

We have implemented the Active Testing framework for the Berkeley UPC [6] compiler. Currently, we support all operations provided by the UPC v1.2 language specification: memory accesses through pointers to shared, bulk transfers (e.g. `upc_memput`), lock operations, and barrier operations. The framework itself is implemented in the UPC programming language and it can be easily ported to other UPC compiler/runtime implementations, such as Cray UPC. We next describe the implementation of the two phases of active testing.

5.1 Implementation of Race Prediction Phase

The runtime instrumentation redefines all memory access and synchronization operations by adding “before” and “after” calls into our analysis framework. For example, for any data access we add THRILLE_BEFORE(type, address) and THRILLE_AFTER(type, address) calls before and after the actual data access statement, respectively. When linking the application with our runtime, a write to shared memory `p[i] = 3` translates into:

```
THRILLE_BEFORE(write, p+i);
upcr_put_pshared_val(p+i, 3);
THRILLE_AFTER(write, p+i);
```

During execution, each task maintains a trace of memory accesses to a particular portion of the shared heap. Whenever a task accesses the shared heap it has to inform the maintainer of that particular region. Overall, during phase one there are two sources of program slowdown: 1) querying a potentially large access trace and; 2) transmitting the query data over the network. In the rest of this section we describe optimizations designed to reduce the overhead of these operations.

Data structures to represent memory accesses: The data structure to represent memory accesses needs to support efficiently both single address queries as well as the address range queries associated with bulk transfers. Previous work on data races focuses on word level memory accesses and uses hash tables to find conflicting addresses. To efficiently find overlapping intervals, we use interval skip lists (IS-list) [26].

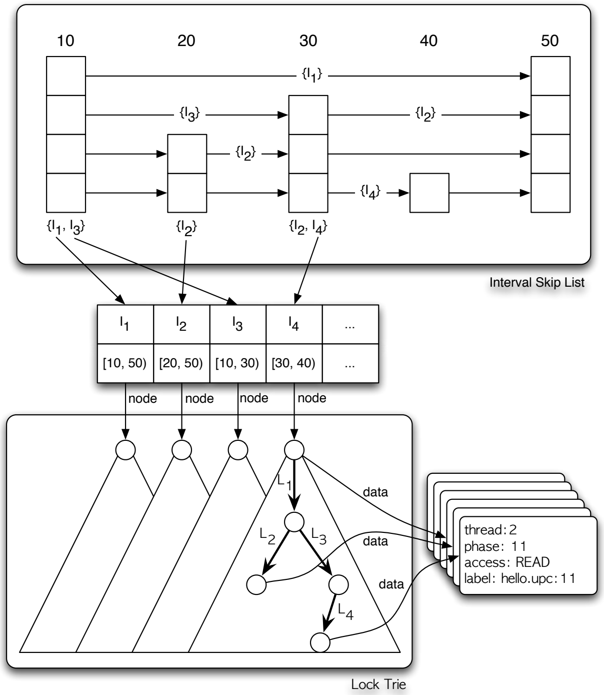


Figure 3: Data structures used to efficiently find weaker-than and racing accesses

Skip lists [51] are an alternative to balanced search trees with probabilistic balancing. They are much easier to implement and perform very well in practice. Skip lists are essentially linked lists augmented with multiple forwarding pointers at each node. The number of forwarding pointers is called the *level* of a node, which is randomly assigned with a geometric distribution when a node is inserted into the list. Higher level nodes are rare, but can skip over many nodes, contributing to the performance of skip lists (expected time $O(\log n)$ for *insert, delete,* and *search*). Skip lists are also space efficient: a node can be stored in memory with an average of four machine words, one word for the key, one word for the pointer to data, and an average of two forward pointers with success probability $p = 0.5$ for the geometric distribution.

An IS-list is essentially a skip list for all the endpoints of intervals it contains, with edges representing the interval spanned by the endpoints. Each node and edge is marked with intervals that cover them. To efficiently handle overlapping queries with minimal space overhead, only the highest level edges that are subintervals of I (i.e. edge $(n_1, n_2) \subseteq I$ and $\not\exists$ edge (n'_1, n'_2). $(n_1, n_2) \subset (n'_1, n'_2) \subseteq I$) need to be marked, and a node n is marked with I if any of the incoming or outgoing edges of that node is marked with I and $n \in I$. For example, in Figure 3, interval $I_2 = [20, 50)$, has markers on the second level forward edge of node 20 and the third level forward edge of node 30, because there are no higher level edges that are contained in I_2.

The operations on IS-lists are time and space efficient. Inserting an interval takes expected time $O(\log^2 n)$ where n is the number of intervals in the IS-list. A search for intervals overlapping a point can be found in expected time $O(\log n + L)$, where L is the number of matching intervals, and searching for intervals overlapping an interval takes ex-

Algorithm 1: FindRace: Find an access in IS-list that races with an access

```
Input: IS-list $isl$ and access $e$
I ← Intervals in $isl$ that overlap with $e.m$;
foreach $interval\ i \in I$ do
    N ← { i.node };
    foreach $node\ n \in N$ do
        if $n.data \neq NULL \wedge n.data.t \neq e.t \wedge n.data.p \,||\, e.p \wedge (n.data.a = WRITE \vee e.a = WRITE)$
        then /* a race is found */
            $e' = (i, \{l{:}\ l \in$ path from $n$ to root$\}, n.data)$;
            return $e'$
        else /* only traverse lock edges not held by $e$ */
            foreach $c \in n.children$ do
                if $c.lock \notin e.l$ then  N ← N ∪ {c};
```

Algorithm 2: AddAccess: Adds an access to IS-list while reporting races and removing stronger accesses

```
Input: IS-list $isl$ and access $e$
if $\nexists e' \in isl\ s.t.\ e' \sqsubseteq e$ then
    if $r$ ← FindRace$(isl, e)$ then
        Report r;
    else
        Insert $e$ into $isl$;
        Remove $\forall e' \sqsupset e$ from $isl$;
```

pected time $O(\log^2 n + L)$. The expected space required for n intervals is $O(n \log n)$.

Figure 3 is an overview of how the database of shared memory accesses is represented. Each memory access event $e = (m, t, l, a, p, s)$ is first grouped by the address range m and inserted into the IS-list. Each interval is associated with a lock trie that represents the locks l held during e. Each node in the trie represents an access with all the locks in the path from the root held. For example, the root of the trie represents an access to m without any locks held. A trie is used to represent locks to efficiently search for accesses racing with e, by only following edges not included in $e.l$. Algorithm 1 shows the full steps for finding racing accesses in the trie.

Query Coalescing: The race prediction phase has to perform remote queries at each individual remote memory access to check for conflicts. On a cluster, this amounts to performing additional data transfers for each transfer present in the application.

In our implementation, each task tracks all the remote accesses locally and delays all the queries until barrier boundaries. Inside barriers, tasks coalesce the query data by memory region into larger messages and perform point-to-point communication with the maintainer of each region. Upon receiving information from all other tasks, each task independently computes all conflicting access that happen within its region of the global shared heap.

Our implementation performs both communication - communication and communication - computation overlap using a "software pipelining" approach. Transfers for query data are asynchronously initiated at a barrier operation and overlapped with each other. These transfers are allowed to proceed until the program executes the next barrier, at which point they are completed, new transfers are initiated and queries are performed for the requests just completed.

Extended Weaker-Than Relation: Keeping track of all shared memory access can incur high space overhead for tasks and increase the amount of communication required between tasks. Thus, we prune redundant information about accesses that do not contribute to finding additional data races. For example, if a task reads and writes to the same memory region, only the write information is required, because any races with the read would also imply a race with the write. Similarly, a wider memory region, an access with less locks held, and a memory region accessed by a lesser number of tasks are all redundant information. Formally, the *weaker-than* relation between memory accesses introduced by Choi et al [10] identifies the accesses that can be pruned. We extend this relation to handle memory ranges.

Definition 5 (Extended Weaker-Than: $\sqsubseteq$). *For two memory access events* e_1 *and* e_2,

$$e_1 \sqsubseteq e_2 \Leftrightarrow e_1.m \supseteq e_2.m \ \wedge \ e_1.t \sqsubseteq e_2.t \ \wedge \ e_1.l \subseteq e_2.l \ \wedge \ e_1.a \sqsubseteq e_2.a$$

where

$$t_i \sqsubseteq t_j \Leftrightarrow t_i = t_j \vee t_i = * \quad (\textit{multiple tasks})$$
$$a_i \sqsubseteq a_j \Leftrightarrow a_i = a_j \vee a_i = WRITE$$

Only the weakest accesses are stored locally and sent to other tasks at barriers. Also, when conflicting races are computed, the weaker-than relation is used in Algorithm 2 to prune redundant accesses from multiple tasks.

Exponential Backoff: We can further prune redundant access information by dynamically throttling instrumentation on statements that are being repeatedly executed. For each static access label (file, line number, variable), we keep a probability for considering these accesses for conflict detection, initially all set to 1.0. Whenever a data race is detected on a statement, we set the probability to 0, effectively disabling instrumentation for that statement for the rest of the execution. Each time an access is recorded, we reduce the probability by a *backoff factor*, eventually disabling instrumentation for this statement after multiple unsuccessful attempts at finding a conflicting access. For our experiments, we used a backoff factor of 0.9 which was a good balance for effectively finding potential data races with low overhead. In Section 6, we discuss the performance gains achieved by these optimizations.

Algorithm 3 is the complete scheduling algorithm for race prediction. The global access list is the communication channel to send shared access information between tasks. Each task also maintains local IS-lists to keep track of its memory accesses separately for each phase and affinity. The probability of considering each program statement is initially set to 1.0. Lines 4-8 are the actions performed for each memory access. We probabilistically add the shared access information to the task's *local* IS-list, while pruning all but the weakest accesses (Algorithm 2). Lines 10-15 handle lock acquires and releases. In case of a notify statement (lines 16-20), before notifying the other tasks (line 19), we make sure that all pending asynchronous transfers are complete (line 17) and initiate asynchronous transfers of accesses in the current phase (line 18). For wait statements (lines 21-37), we first wait for all other tasks (line 22), and then initiate asynchronous transfers of accesses in the current phase (line 23). In lines 24-36, we check for barrier aware potential data

Algorithm 3: ThrilleRacerScheduler

```
Initially,
Global: $\forall t \in \mathcal{T}, p \in \mathbb{N}, o \in \mathcal{T}.\ global\_acc\_list[t,p,o] = \emptyset$
Task local: $p = 0,\ L = \emptyset,\ \forall s.\ prob[s] = 1.0$, and
            $\forall p \in \mathbb{N}, o \in \mathcal{T}.\ islist[p,o] = \emptyset$
while $i$ := next instruction of task $t$ do
    switch $i$ do
        case $i$ = MEM(m,a,s)
            e ← (m,t,L,a,p,s);
            if random() < prob[s] then
                AddAccess(islist[p,owner(m)], e);
                prob[s] *= BACKOFF;
            Execute $i$;
        case $i$ = LOCK(l)
            Execute $i$;
            $L \leftarrow L \cup l$;
        case $i$ = UNLOCK(l)
            $L \leftarrow L \backslash l$;
            Execute $i$;
        case $i$ = UPC_NOTIFY
            Synchronize all pending transfers;
            foreach $t' \neq t$ do Asynchronously send
            $islist[p,t']$ to $global\_acc\_list[t,p,t']$;
            Execute $i$;
            $p$++;
        case $i$ = UPC_WAIT
            Execute $i$;
            foreach $t' \neq t$ do Asynchronously send
            $islist[p,t']$ to $global\_acc\_list[t,p,t']$;
            /* Check for races among all accesses
               in phase $p-2$                              */
            foreach $t' \neq t$ do
                foreach $e \in global\_acc\_list[t',p-2,t]$ do
                    AddAccess(islist[p-2,t], e);
            /* Check races in phases p-2, p-3    */
            foreach $e \in islist[p-2,t]$ do
                AddAccess(islist[p-3,t], e);
            /* Check races in phases p-2, p-1    */
            foreach $e \in islist[p-2,t]$ do
                AddAccess(islist[p-1,t], e);
            foreach $t' \neq t$ do
                foreach $e \in global\_acc\_list[t',p-1,t]$ do
                    AddAccess(islist[p-1,t], e);
            $p$++;
        otherwise
            Execute $i$;
```

race pairs (Definition 4) in the previous barrier phases based on local information (islist) and information received from other tasks (global_acc_list).

5.2 Implementation of Race Confirmation Phase

After collecting potential data race pairs from phase I, we run the race testing phase on each pair to observe the effects of the race in isolation. Algorithm 4 shows the complete scheduling algorithm. We use an exponential back-off optimization similar to phase I to keep the overhead of phase II low and achieve scalability. `bupc_sems`, an extension in the Berkeley UPC runtime, are used as semaphores to control the execution order of tasks. Each task announces the memory access it is currently pending on in the global

Algorithm 4: ThrilleTesterScheduler

```
   Input: Potential race pair $s_1, s_2$
1  Initially,
   Global: $Prob = 1.0, \forall t \in \mathcal{T}.\ sem[t] = 1,\ pending[t] =$
           $NULL$
2  while $i$ := next instruction of task $t$ do
3      if $Prob > 0 \wedge i$ = MEM(m,a,s) then
4          if $s = s_1 \vee s = s_2$ then
5              if $\exists t'.\ pending[t'].m \cap m \neq \emptyset \wedge$
               $(pending[t'].a = WRITE \vee a = WRITE) \wedge$
               $pending[t'].s \neq s$ then
6                  Report race between tasks t and t';
7                  sem[t'].signal();
8                  Prob = 0;
9              random() < Prob pending[t] = (m,a,s);
10             sem[t].wait(TIMEOUT);
11             pending[t] = NULL;
12             Prob *= BACKOFF;
13     Execute $i$;
```

data structure *pending*. Here, we only need to consider the shared memory accesses in the program (line 3). If the statement label matches one of the statements in the potential data race pair (line 4), we first check if any other task is pending on the other statement (line 5). If one is found to be pending on an access with an overlapping memory address and either access is a write, we report a real race and signal the other task to proceed. Once a real race is found, we disable testing (line 8) and continue execution of the program normally.

If no other pending task meeting the race criteria is found, we probabilistically post the information about the access (line 10) and pause the task for some time (line 11). At line 12, either a real race was found and the semaphore released by the other task (line 7) or the timeout could have expired. In either case, we no longer announce this task as pending on the memory access and reduce the probability of pausing.

Phase II is not guaranteed to be a complete approach—it cannot confirm all real data races. However, the approach was able to confirm all previously known races and artificially injected ones in our benchmarks.

6. EVALUATION

We evaluate data race detection on UPC fine-grained and bulk communication benchmarks. For implementations using bulk communication primitives we use the NAS Parallel Benchmarks (NPB) [43, 44], releases 2.3, 2.4, and 3.3. For lack of space, we do not discuss NPB implementation details. The fine-grained benchmarks were written by researchers outside of our group and reflect the type of communication/synchronization that is present in larger applications during data structure initializations, dynamic load balancing, or remote event signaling.

The *guppie* benchmark performs random read/modify/write accesses to a large distributed array, a common operation in parallel hash table construction. The amount of work is static and evenly distributed among tasks at execution time. The *mcop* benchmark solves the matrix chain multiplication problem [13]. This is a classical combinatorial problem that captures the characteristics of a large class of parallel dynamic programming algorithms. The matrix data is distributed along columns, and communication occurs in the form of accesses to elements on the same row. The *psearch* benchmark performs parallel unbalanced tree search [48]. The benchmark is designed to be used as an evaluation tool for dynamic load balancing strategies.

We built UPC-Thrille using the BUPC release 2.12.2 with gcc 4.5.2.

6.1 UPC-Thrille on Shared Memory Systems

In Table 1 we show a summary of results obtained for UPC-Thrille when running the benchmarks on a quad-core 2.66GHz Intel Core i7 workstation with 8GB RAM. We report the total lines of source code (LoC) for each benchmark and the runtime of the original program without any analysis. We report the runtime (RT) and overhead(OH) of the program with race prediction (phase I) enabled and the average runtime and overhead for program re-execution with race confirmation (phase II).

The total number of potential races predicted by phase I is reported in column six. This number reflects the number of racing memory accesses performed throughout the application execution. The number in parentheses represents the unique pairs of racing statements reported by UPC-Thrille that are associated with the runtime races. Each pair of racing statements identified in phase I is tested in phase II and we report the number of confirmed races in the last column.

The race prediction phase added for most benchmarks only a small runtime overhead of up to 15%. The overhead of the race confirmation phase is determined by the granularity of the delays (pauses) introduced in the thread schedule, as well as by the dynamic count of such pauses. For all experiments, the overhead of phase II was negligible when using a delay of 10ms.

Our results demonstrate that UPC-Thrille is able to precisely detect and report the races present in the benchmarks evaluated. For all these benchmarks, the races manifest regardless of the concurrency of the execution or the actual input set. Any testing run at any concurrency will uncover the same set of races.

6.2 Races Found

guppie: These races are expected in the program, as random updates are made to a global table. One race is between a read of a table entry from one task and a write to the same entry from another task. The other race is between two writes from different tasks to the same location in the table.

knapsack: This is our example program in Figure 1. Through active testing, we can confirm that both races are indeed real. Furthermore, by controlling the order in which the race is resolved, we can force the program into an error. If the initial value at line 33 (or 34) is read before the write at line 142 (line 143, respectively), then the verification check of the solution fails.

psearch: The races are in code that implements work stealing. Shared variables hold the amount of stealable work available for each task. A real race can result in work stealing to fail, but does not affect the correctness of the program because of carefully placed mutexes. One of the predicted races is unrealizable because of this custom synchronization.

NPB 2.4 FT: This race is real but benign, as all tasks initialize the variable `dbg_sum` to 0 at the same time.

NPB 2.4 MG: Two shared variables are read by each task and then reset to 0 by task 0. It seems highly suspicious that there is no barrier to wait for all the reads. We reproduced both the races but it did not affect the solution computed. After inspecting the code, the variables are actually used only by task 0 for reporting purposes.

NPB 2.3 FT: The races in this version is quite devious. The accesses to a shared variable `dbg_sum` is protected by a lock `sum_write`. Then how could the accesses be racing? It turns out that each task holds a different lock, because the wrong function `upc_global_lock_alloc()` was called to allocate the lock — a different global lock is returned to each calling task. `upc_all_lock_alloc()`, a collective function that returns the same lock to all tasks, should have been used instead.

NPB 3.3 BT and SP: The races predicted by phase I in these benchmarks cannot be confirmed in phase II, because these benchmarks use custom synchronization operations. Both benchmarks implement point-to-point synchronization operations where one task performs a write operation (`write a`) followed by a `upc_fence` (null strict memory reference), while another task is polling on the value of the variable `a`. The false positives are not caused by the strict operation per se but by lack of semantic information about the application.

6.3 Scalability of UPC-Thrille on Distributed Memory Systems

Some applications might have races that occur only at certain concurrency levels; thus, the overall scalability of UPC-Thrille is important. In Figure 4 we present results for NPB class C scaled up to 256 cores and class D up to 1024 cores on the the Franklin [25] Cray XT4 system at NERSC. The nodes contain a quad-core AMD Budapest 2.3GHz processor, connected with a Portals interconnect. The latency for this network using Berkeley UPC [6] is around 11μs for eight byte messages.

For all benchmarks, we plot the original application speedup (*no analysis*) and the speedup with race prediction enabled (*phase 1*). For benchmarks with races predicted, we also present the scalability of the confirmation phase (*phase 2*). The average slowdown of both phase I and phase II are less than 1%. The maximum slowdown observed for phase I was 8.1% for IS class C at 128 cores. For phase II, MG class D at 1024 cores had the maximum slowdown of 15%. These results are obtained with an exponential backoff of 0.9.

For most of the benchmarks, the race prediction and confirmation phases scale well. For phase I, our implementation introduces overhead mostly in barrier operations. Figure 5 compares the average per-barrier overhead of active testing with the average application inter-barrier time, noted as *barrier length*. The *barrier length* is computed as the total application original execution time divided by the number of barriers. The average per-barrier overhead of active testing is computed as the total overhead divided by the number of barriers and we further sub-divide it into computation and communication overhead. *serial computation* is the average overhead of the IS-lists lookup while *communication* is the average overhead of communication added by UPC-Thrille. Our implementation overlaps UPC-Thrille specific commu-

				ThrilleRacer			ThrilleTester		
Benchmark		LoC	Runtime	RT	OH	Pot. races	Avg.RT	Avg.OH	Conf. races
guppie		277	2.094s	2.346s	1.120x	157 (2)	2.129s	1.017x	2
knapsack		191	2.099s	2.412s	1.149x	2 (2)	2.136s	1.018x	2
laplace		123	2.101s	2.444s	1.163x	0 (0)	-	-	-
mcop		358	2.183s	2.198s	1.007x	0 (0)	-	-	-
psearch		777	2.982s	3.037s	1.018x	11 (3)	3.095s	1.038x	2
NAS Parallel Bench.	FT 2.3	2306	8.711s	9.243s	1.061x	25 (2)	9.131s	1.048s	2
	CG 2.4	1939	3.812s	3.831s	1.005x	0 (0)	-	-	-
	EP 2.4	763	10.022s	10.109s	1.009x	0 (0)	-	-	-
	FT 2.4	2374	7.036s	7.045s	1.001x	6 (1)	7.334s	1.042x	1
	IS 2.4	1449	3.073s	3.106s	1.011x	0 (0)	-	-	-
	MG 2.4	2314	4.895s	5.045s	1.031x	9 (2)	4.955s	1.012x	2
	BT 3.3	9626	48.78s	49.04s	1.005x	40 (8)	49.15s	1.008x	0
	LU 3.3	6311	37.05s	37.22s	1.005x	0 (0)	-	-	-
	SP 3.3	5691	59.56s	59.70s	1.002x	32 (8)	61.36s	1.030x	0

Table 1: Results for race-directed active testing on benchmarks on a 4 core workstation. We present results for NPB class A.

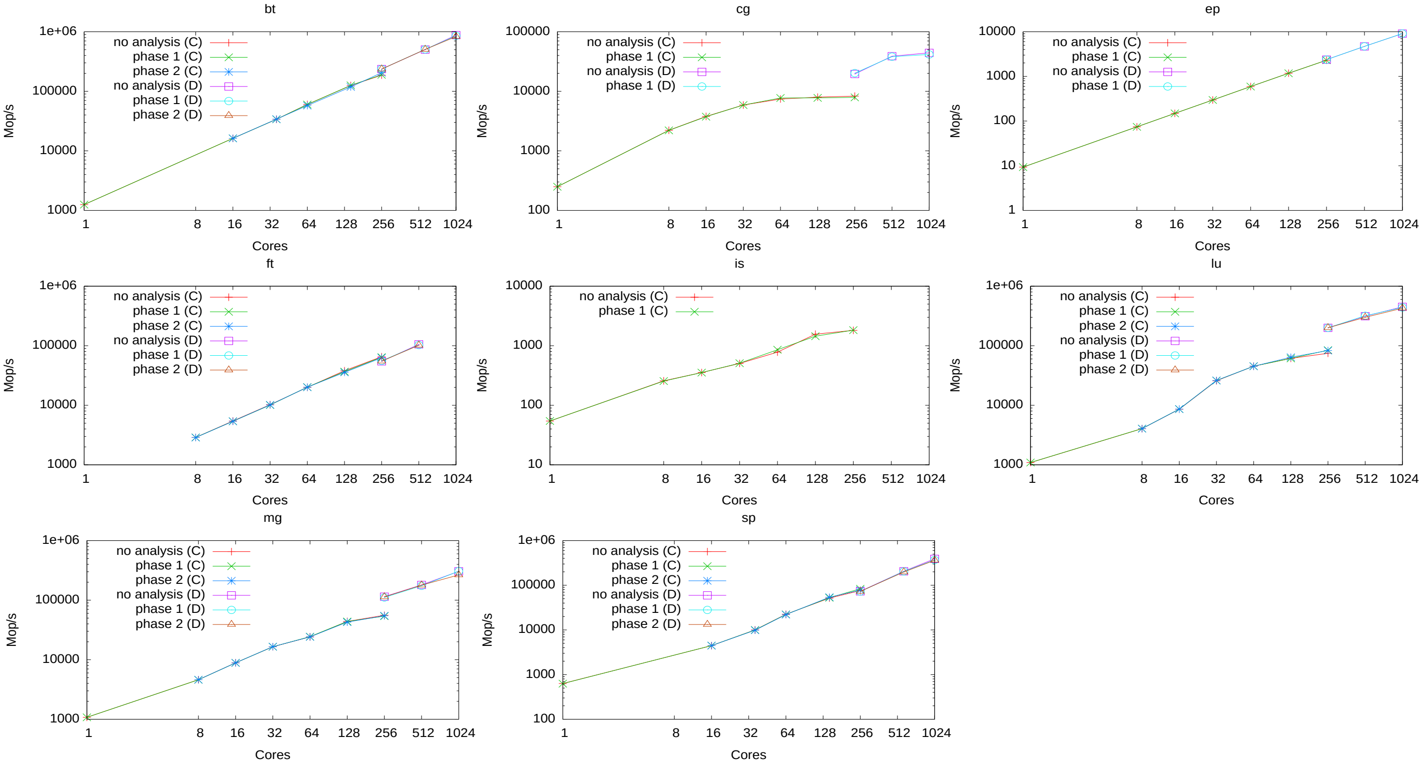


Figure 4: Scalability of active testing up to 1024 cores for NPB classes C and D. Class D was not available for benchmark IS. We were unable to run FT class C on 1 core and class D on 1024 cores.

nication with both internal lookups and the whole computation already present in the application between barriers. Thus, *communication* measures only residuals after overlapping and in most cases lookups are the main cause of slowdown.

For reference, CG class C running on 256 cores (CG-C-256) executes on average a barrier every 335μs, CG-D-256 every 2.7ms, while CG-D-1024 executes one at 1ms intervals. In all these cases, the race prediction phase has a low overhead, accounting for a few percent slowdown. These considerations indicate that active testing has a good scalability potential.

All optimizations applied provide good performance benefits. *weaker-than* and the exponential back-off reduce the volume and frequency of communication operations. For example, in MG class C running on 64 cores, there were a total of 4.7M shared accesses (memput, memget) for all tasks. With *weaker-than*, we pruned 2.1M accesses (46%), mostly shared reads because writes are weaker than reads (Definition 5). With exponential backoff (factor=0.9), we further pruned to 53K accesses but were still able to predict all the data races as before. Each task communicated an average of 10 bytes per barrier. The maximum bytes sent by a task at a barrier was 23KB, but this number went down after the effects of dynamic throttling kicked in.

7. OTHER RELATED WORK

Dynamic techniques for finding concurrency bugs can be classified into two classes: predictive techniques and precise techniques Predictive dynamic techniques [54, 63, 10, 46, 2, 27, 5, 34, 64, 65, 22, 20, 21] could predict concurrency bugs that did not happen in a concurrent execution; however, such techniques can report many false warnings. Phase I of our active testing is a predictive technique. Precise dynamic techniques, such as happens-before race detection [55, 16, 1, 11, 38, 53, 12, 42, 23] and atomicity monitoring [67, 37, 19, 24], are capable of detecting concurrency bugs that actually happen in an execution. Therefore, these techniques are precise, but they cannot give good coverage as predictive dynamic techniques.

More recently, a few techniques have been proposed to confirm potential bugs in concurrent programs using random testing. Active random testing [56, 49, 34, 33] has been proposed to confirm data races, deadlocks, and atomicity violations by biasing a random model checker. Havelund et al. [4] uses a directed scheduler to confirm that a potential deadlock cycle could lead to a real deadlock. Similarly, ConTest [45] uses the idea of introducing noise to increase the probability of the occurrence of a deadlock. Shacham et al. [57] have combined model checking with lockset based algorithms to prove the existence of real races. CTrigger [50] uses trace analysis, instead of trying out all possible schedules, to systematically identify (likely) feasible unserializable interleavings for the purpose of finding atomicity violations. SideTrack [69] improves monitoring for atomicity violation by generalizing an observed trace.

Static verification [17, 29, 52, 7, 41, 66] and model checking [18, 31, 28, 61, 68] or path-sensitive search of the state space is an alternative approach to finding bugs in concurrent programs. Model checkers, being exhaustive in nature, can often find all concurrency related bugs in concurrent programs. Unfortunately, model checking does not scale with program size. CHESS [39, 40] tames the scalability problem of model checking by bounding the number of preempting context switches to small numbers. However, CHESS is not directed towards finding common concurrency bugs quickly—it is geared towards systematic search and better coverage.

So far there have been a lot of research effort to verify and test concurrent and parallel programs written in Java and C/pthreads for non-HPC platforms; the huge body of literature listed above supports this fact. There have also been effort to test and verify HPC programs, mostly focussed on C/MPI programs. ISP [59] is a push-button dynamic verifier capable of detecting deadlocks, resource leaks, and assertion violations in C/MPI programs. DAMPI [62] overcomes ISP's scalability limitations and scales to thousands of MPI processes. Like ISP, DAMPI only tests for MPI Send/Recv interleavings, but runs in a distributed way. In contrast to our work, DAMPI instruments and reasons only about the ordering of Send/Recv operations with respect to the MPI ranks, and not about the memory accessed by these operations. Both ISP and DAMPI assume that program input is fixed. TASS [58] removes this limitation by using symbolic execution to reason about all possible inputs to a MPI program, but it is work only at inception. MPI messages can be intercepted and analyzed for bugs and anomalies. Intel MessageChecker [14] does a post-mortem analysis after collecting message traces, while MARMOT [35] and Umpire [60] check at runtime. Our proposed active testing technique targets finding memory bugs in HPC programs and has to extend the previous approaches with techniques to reason about local memory accesses in conjunction with communication operations.

8. CONCLUSION

While a lot of work has been performed for bug finding in concurrent programs, little has permeated into the HPC application domain. Most of the previous HPC work targets C/MPI programs and reasons about interleavings of MPI Send/Recv operations with respect to each other and ignoring any intervening memory accesses. We are the first to develop a race directed testing tool for distributed memory parallel programs. Out tool integrates reasoning about memory accesses and communication operations. We extend previous formalisms to handle HPC specific programming model constructs and address several scalability challenges introduced by the distributed memory nature of these applications. Our initial results show scalability up to a thousand cores.

9. ACKNOWLEDGEMENTS

Significant portions of the first author's work were performed while on internship at LBNL. This research is supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906, CCF-0747390, CCF-1018729, and CCF-1018730, and by an Alfred P. Sloan Foundation Fellowship. Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

10. REFERENCES

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *18th International Symposium on*

Computer architecture (ISCA), pages 234–243. ACM, 1991.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *20th International Conference on Automated software engineering (ASE)*, pages 233–242. ACM, 2005.

[3] A. Aiken and D. Gay. Barrier inference. In *Principles of programming languages (POPL)*, pages 342–354, New York, NY, USA, 1998. ACM.

[4] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD'06*, pages 41–50, 2006.

[5] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In *Parallel and Distributed Systems: Testing and Debugging 2005 (PADTAD'05)*, 2005.

[6] Berkeley UPC. Available at http://upc.lbl.gov/.

[7] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[8] J. Burnim, C. Stergiou, and K. Sen. Testing concurrent programs on relaxed memory models. In *International Symposium on Software Testing and Analysis*, 2011.

[9] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, and K. W. E. Brooks. Introduction to UPC and language specification, 1999.

[10] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming language design and implementation (PLDI)*, pages 258–269, New York, NY, USA, 2002. ACM.

[11] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[12] M. Christiaens and K. D. Bosschere. TRaDe, a topological approach to on-the-fly race detection in java programs. In *JavaTM Virtual Machine Research and Technology Symposium*, pages 15–15. USENIX, 2001.

[13] T. Cormen, C. Leiserson, and R. Rivset. *Introduction to Algorithms.* The MIT Press, 1994.

[14] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Software engineering for high performance computing system applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM.

[15] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid parallel programming with MPI and Unified Parallel C. In *Computing frontiers (CF)*, CF '10, 2010.

[16] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Workshop on Parallel and Distributed Debugging*, 1991.

[17] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[18] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking.* MIT Press, 1999.

[19] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, pages 52–65. Springer, 2008.

[20] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

[21] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Computer Aided Verification (CAV)*, pages 248–262. Springer-Verlag, 2009.

[22] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *31st Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[23] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Programming language design and implementation (PLDI).* ACM, 2009.

[24] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Programming language design and implementation (PLDI)*, pages 293–303. ACM, 2008.

[25] Franklin: Cray XT4. At http://www.nersc.gov/users /computational-systems/franklin/.

[26] E. N. Hanson and T. Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Workshop on Algorithms and Data Structures*, pages 153–164. Springer, 1992.

[27] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *7th International SPIN Workshop on Model Checking and Software Verification*, pages 245–264, 2000.

[28] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[29] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. *SIGPLAN Not.*, 39(6):1–13, 2004.

[30] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Technical report, Berkeley, CA, USA, 2001.

[31] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[32] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *Foundations of Software Engineering (FSE).* ACM, 2010.

[33] P. Joshi, M. Naik, C.-S. Park, and K. Sen. An extensible active testing framework for concurrent programs. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, 2009.

[34] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Programming Language Design and Implementation (PLDI)*, 2009.

[35] B. Krammer, M. Müller, and M. Resch. Runtime checking of MPI applications with MARMOT. In *Mini-Symposium Tools Support for Parallel Programming, ParCo 2005, Malaga, Spain, September 12 - 16, 2005.*, 2005.

[36] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[37] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. *SIGARCH Comput. Archit. News*, 34(5):37–48, 2006.

[38] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, pages 24–33. ACM, 1991.

[39] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Programming Language Design and Implementation (PLDI)*, 2007.

[40] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Operating Systems Design and Implementation (OSDI)*, pages 267–280, 2008.

[41] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[42] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Language Design and Implementation (PLDI)*, pages 22–31, 2007.

[43] The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB.

[44] The UPC NAS Parallel Benchmarks. Available at http://upc.gwu.edu/download.html.

[45] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From exhibiting to healing. In *8th Workshop on Runtime Verification*, 2008.

[46] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.

[47] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and practice of parallel programming (PPoPP)*, pages 167–178. ACM, 2003.

[48] S. Olivier and J. Prins. Scalable dynamic load balancing using UPC. In *International Conference on Parallel Processing (ICPP)*, ICPP '08, 2008.

[49] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Foundations of Software Engineering (FSE)*. ACM, 2008.

[50] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Architectural support for programming languages and operating systems (ASPLOS)*, pages 25–36. ACM, 2009.

[51] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[52] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *Programming language design and implementation (PLDI)*, pages 14–24. ACM, 2004.

[53] M. Ronsse and K. D. Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[54] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[55] E. Schonberg. On-the-fly detection of access anomalies. In *Programming Language Design and Implementation (PLDI)*, volume 24, pages 285–297, 1989.

[56] K. Sen. Race directed random testing of concurrent programs. In *Programming Language Design and Implementation (PLDI)*, 2008.

[57] O. Shacham, M. Sagiv, and A. Schuster. Scaling model checking of dataraces using dynamic information. *J. Parallel Distrib. Comput.*, 67(5):536–550, 2007.

[58] S. F. Siegel and T. K. Zirkel. Automatic formal verification of MPI-based parallel programs. In *Principles and practice of parallel programming*, PPoPP '11, pages 309–310, New York, NY, USA, 2011. ACM.

[59] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: a tool for model checking MPI programs. In *Principles and practice of parallel programming*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM.

[60] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[61] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Engineering (ASE)*. IEEE, 2000.

[62] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *Supercomputing*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[63] C. von Praun and T. R. Gross. Object race detection. In *Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 70–82. ACM, 2001.

[64] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *3rd Workshop on Run-time Verification*, volume 89 of *ENTCS*, 2003.

[65] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 137–146. ACM Press, Mar. 2006.

[66] A. Williams, W. Thies, and M. Ernst. Static deadlock detection for Java libraries. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 602–629, 2005.

[67] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[68] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *SPIN*, Lecture Notes in Computer Science, pages 288–305. Springer, 2008.

[69] J. Yi, C. Sadowski, and C. Flanagan. SideTrack: generalizing dynamic atomicity analysis. In *7th Workshop on Parallel and Distributed Systems*, pages 1–10. ACM, 2009.

[70] Y. Zhang and E. Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Principles and practice of parallel programming*, PPoPP '07, 2007.

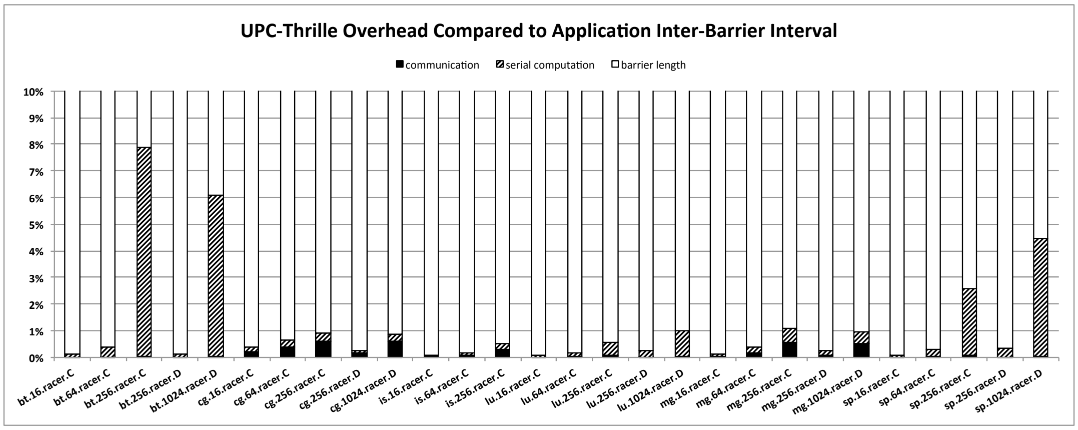


Figure 5: Communication and computation overhead of active testing phase 1 in NPB