Thread Contracts for Safe Parallelism *

Rajesh K. Karmani P. Madhusudan Brandon M. Moore

University of Illinois at Urbana-Champaign
{rkumar8,madhu,bmmoore}@illinois.edu

Abstract

We build a framework of thread contracts, called ACCORD, that allows programmers to annotate their concurrency co-ordination strategies. ACCORD annotations allow programmers to declaratively specify the parts of memory that a thread may read or write into, and the locks that protect them, reflecting the concurrency co-ordination among threads and the reason why the program is free of data-races. We provide *automatic* tools to check if the concurrency co-ordination strategy ensures race-freedom, using constraint-solvers (SMT solvers). Hence programmers using ACCORD can both formally state and prove their co-ordination strategies ensure race freedom.

The programmer's implementation of the co-ordination strategy may however be correct or incorrect. We show how the formal ACCORD contracts allow us to automatically insert *runtime* assertions that serve to check, during testing, whether the implementation conforms to the contract.

Using a large class of data-parallel programs that share memory in intricate ways, we show that natural and simple contracts suffice to document the co-ordination strategy amongst threads, and that the task of showing that the strategy ensures race-freedom can be handled efficiently and automatically by an existing SMT solver (Z3). While co-ordination strategies can be proved race-free in our framework, failure to prove the co-ordination strategy race-free, accompanied by counter-examples produced by the solver, indicates the presence of races. Using such counterexamples, we report hitherto undiscovered data-races that we found in the long-tested `applu_l` benchmark in the Spec OMP2001 suite.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—Parallel programming; D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification—Programming by contract; F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs

General Terms Reliability, Verification

Keywords concurrent contracts, constraint solvers, data-races, testing

1. Introduction

Perhaps the most generic error in shared-memory concurrent programs is that of a data-race— two concurrent accesses to a memory location, where at least one of them is a write. The primary reason why data-races must be avoided is that memory models of many high-level programming languages do not assure sequential consistency in the presence of data-races [16]. In some cases, like the new semantics of *C++*, the semantics of programs is *not even defined* when data-races are present [5, 7]. Consequently, data-races in high-level programming languages are almost always (if not always) symptomatic of bugs in the program, and are best avoided by mainstream programmers. However, there is no really acceptable shared-memory programming paradigm today that allows programmers to write certifiably race-free code.

***ACCORD*:**
While there have been several attempts to build new shared-memory programming languages that avoid races by design (see Guava [4] and DPJ [6] for example), we propose in this work to instead work with the languages that are currently available, but provide an *annotation framework*, similar to code contracts for sequential programs [17, 18], that allows a programmer to express the concurrency co-ordination strategy amongst threads, and prove them race-free.

Intuitively, a programmer's attempt to parallelize a piece of sequential code consists of two phases:

(a) to come up with a *co-ordination strategy* of the computation amongst threads (for example, deciding what parts of the memory each thread will work on, the locks they will employ to have mutually exclusive access to data, etc.), and

(b) to *implement* the co-ordination strategy in terms of real code.

Unfortunately, the results of the first phase often remain undocumented. A diligent programmer may spell out the strategy in comments written in a natural language, but often the strategy is lost in the code.

In this paper, we propose a *formal annotation language* called ACCORD (Annotations for Concurrent Co-ORDination), using which the programmer can express the concurrent co-ordination strategy. We postulate that doing this has multiple benefits in ensuring correctness. First, we propose automatic solutions to the problem of checking whether the strategy is correct (i.e. ensures race-freedom). Second, we propose automatic insertion of assertions in a program that can be used during testing to check if the program conforms to the annotated co-ordination strategy.

The philosophy behind ACCORD specifications is to document *memory-access contracts* between threads. When several threads are spawned at a point, an ACCORD annotation expresses two aspects of the sharing strategy: (a) the set of memory locations that each thread T will read and write to *without* mutexes such as locks, and (b) the memory locations that each thread T will

* This work was funded partly by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign, sponsored by Intel Corp. and Microsoft Corp., and NSF Career Award #0747041.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0119-0/11/02. . . $10.00

access only when possessing an associated lock. These read/write regions can be *conservative*, in that the programmer can specify a superset of the regions actually touched, as long as the annotations are sufficient to argue that the program is race-free. The key idea is that such thread contracts are extremely natural to write (indeed, the programmers clearly know this when they come up with the co-ordination strategy, even before they write the code), and moreover, is sufficient to argue and establish that the strategy ensures race-freedom.

In this paper, we are primarily concerned with highly parallel programs written for speed (as opposed to programs that are inherently concurrent, like client-server programs). Examples of popular shared-memory programming languages in this domain are OPENMP, CILK, and TBB. The ACCORD annotation framework works for a restricted class of *array-based* parallel programs that employ fork-join parallelism. Such programs are the most commonly used paradigm for writing shared-memory parallel programs. For instance, OPENMP primarily allows fork-join parallelism and most programs written in OPENMP tend to work by partitioning arrays in intricate ways to exploit parallelism, and moreover the forked threads seldom *communicate* in any way until the join-point. The restriction to fork-join parallelism allows for a much simplified syntax of annotations to express the co-ordination strategy that is both intuitive as well as easy to write.

ACCORD specifications are written both at *forking* points of threads as well as *function boundaries*. Annotations at function boundaries describe also the read and write regions and the locking strategy in the function, and are useful for modular reasoning and testing.

Verifying co-ordination strategies: The idea of writing ACCORD annotations is that it captures formally the concurrency co-ordination strategy between threads, and hence is amenable to analysis. In particular, an important question to ask is whether the annotated co-ordination strategy is adequate to ensure race-freedom; we call this the *annotation adequacy check*.

Our primary contribution in this paper is to show that annotation adequacy for many array-based fork-join programs can be proved automatically using *constraint solvers*. Intuitively, proving whether the co-ordination strategy ensures race-freedom reduces to the problem of verifying whether two threads are allowed by the strategy to simultaneously access a variable, with one of the accesses being a write. This can be compiled into a constraint satisfaction problem (often using integer arithmetic constraints, uninterpreted functions, and array theories) and hence can be handled by constraint solvers based on SMT-solving technology (such as the solver Z3, from Microsoft Research [10]). Furthermore, consistency checks of various ACCORD annotations at different points in the program (such as checking if the read/write sets of an parallel loop is a superset of the read/write sets of an inner loop or a function called within) can also be posed as constraints that state the inclusion of regions, and can be verified using an SMT solver.

Expressing the co-ordination strategy formally and using automatic static-checking to show that the strategy maintains concurrency safety and is internally consistent goes a long way in writing correct programs.

Compiling co-ordination strategy to assertions for testing: The documentation of the co-ordination strategy using a formal syntax also serves as a *specification* of the code implementing the strategy. We show in this paper that we can automatically transform an annotated parallel program to one with assertions that tests whether the threads conform to the co-ordination strategy expressed as ACCORD annotations. The function-level annotations allow us to modularly check the ACCORD annotations at runtime against the local regions defined for the function (given that the consistency of the annotations has been checked). This allows us to check whether the program meets the co-ordination strategy during the *testing* of the program on test inputs. Any violations during such tests would indicate that the co-ordination strategy hasn't been implemented correctly in the program.

***Experience and Evaluation*:** We have applied our framework over several non-trivial *data-parallel algorithms* that employ fork-join concurrency with extremely intricate sharing of data.

The first suite of programs include simple matrix multiplication routines, a race-free implementation of quicksort that uses parallel rank computation to partition an array with respect to a pivot, the successive over-relaxation (SOR) program that uses an intricate checker-board separation of a 2D-array combined with barriers, and an LU Factorization program. Our second suite includes benchmarks from the Java Grande Forum benchmark suite [27]. This includes a Montecarlo algorithm that uses locks as well as separation to achieve race-free parallelism and sparse matrix multiplication, where the concurrent sharing strategy is dynamically determined using *indirection arrays*. We also include in our suite buggy variants of the above correct programs, which at first glance seem correct but have subtle errors.

On a third suite of benchmarks, we examine large programs written in OPENMP and annotate them to prove race-freedom. These examples comprise thousands of lines of code, but require very little annotation (at most 37 for a program), and are sufficient to capture the co-ordination strategy in sufficient detail to argue why it ensures race-freedom. While annotating these programs, we found one that couldn't pass our verification tools, which led us to find subtle data-races in a SPEC OMP2001 benchmark called `applu_l` that has escaped detection for more than 10 years (the NAS benchmark suite, however, seems to have a correct version).

Our experience has been that ACCORD specifications are extremely *natural* to write by the programmer (we wrote annotations for large programs that we didn't even write), and it captures the sharing strategy directly using a simple declarative language. Furthermore, the automatic constraint-solvers such as Z3 can easily and efficiently prove that the co-ordination strategies imply race-freedom. As programmers, we found we gained considerable confidence that the program is free of data-races by knowing the co-ordination strategies ensured race-freedom and that the program conformed to the co-ordination strategy in the testing phase.

In summary, our proposal is that programmers formally write, using a succinct declarative notation, the co-ordination strategy they are implementing. Doing this allows the effective use of analyses tools: one that proves that the co-ordination strategy indeed does ensure race-freedom, and the other that, during the testing phase, checks if the program implements the co-ordination strategy. We believe that this greatly helps in writing race-free code, and through experience on annotating a large suite of programs, we show that correct programs have succinct annotations that can be proved to ensure race-freedom, and that failure to provide verifiable annotations leads to finding bugs.

2. An Illustrative Example

In this section, we provide an overview of ACCORD approach using a simple example. Listing 1 shows the implementation of parallel matrix multiplication algorithm in an imperative, C-like language (ignoring the `reads`, `writes`, `where` clauses for now). The program takes two matrices $A[m,n]$ and $B[n,p]$ as input, and computes the product as $C[m,p]$.

ACCORD ***contract for read and write sets*** Lines 3-5 and 8-9 are ACCORD annotations. The annotation has a `reads` clause (line 8) and a `writes` clause (line 9) that declaratively specify, for every parallel iteration of the `foreach` statement at line 7, the set of memory locations that are read and written by the iteration, respectively. For each iteration k, the read and write annotations specify the locations that the thread corresponding to iteration k will access. Note that the annotation is allowed to refer to all program variables that are in scope. This feature allows a simple and succinct annotation for this program.

Listing 1 Fully parallel implementation of matrix multiplication

```
1  void mm (int[m,n] A, int[n,p] B) {
2    foreach (int i := 0; i < m; i:=i+1)
3      reads A[i,$x],B[$x,$y],C[i,$y] writes C[i,$y]
4      where 0 <= $x and $x < n
5          and 0 <= $y and $y < p  {
6      for (int j := 0; j < n ; j:=j+1)
7        foreach (int k := 0; k < p; k:=k+1)
8          reads A[i,j],B[j,k],C[i,k]
9          writes C[i,k] {
10         C[i,k] := C[i,k] + (A[i,j] * B[j,k]);
11 } }    }
```

Observe that the annotation at lines 3–5 also introduce *auxiliary variables* ($\$x$ and $\$y$); these are *implicitly universally quantified.* This annotation also demonstrates the usage of a `where` clause. A `where` clause constrains (specifies bounds for) the variables in the `reads` and `writes` clauses. This annotation declares that the parallel computation corresponding to index i can read $A[i, \$x]$, $B[\$x, \$y]$ and $C[i, \$y]$, and write to $C[i, \$y]$, *for any values of* $\$x$ *and* $\$y$ that satisfy the condition: $0 \leq \$x < n$ and $0 \leq \$y < p$.

Notice that the annotation for this program is a simple and natural way to declare the co-ordination (sharing strategy) among threads, and indeed, the programmer ought to have this separation of memory in mind when reasoning about race-freedom. While the sharing strategy in this program is not hard (and static analysis tools can synthesize them for this example), the strategy is often very complex in larger examples: the regions can be complex (defined using complex arithmetical and logical constraints), they can be dictated by data stored in arrays (like in parallelism using indirection arrays), can be predicated by conditions that change the sharing strategy (for example, code may decide to handle certain instances using parallelism while others are handled sequentially when there is little parallelism to exploit), etc. Moreover, in order to see that the sharing strategy ensures race-freedom, we may *require* properties of data computed just before the fork-point: for example, in programs that parallelize using indirection arrays, the annotation may say that each thread i writes to $B[A[i]]$, and we may need to know that the array $A[]$ has *distinct* values in order to realize that the forked threads will not cause races. The latter condition is captured using a `requires` clause in the ACCORD annotations.

Proving the co-ordination strategy ensures race-freedom (annotation adequacy) We *automatically* generate a logical formula from the annotation (not the program) that serves as a verification condition for checking whether the co-ordination strategy ensures race-freedom. Specifically, the formula is a conjunction of two subformulas: (a) there exist no two threads such that one of them reads and the other writes to the same location, and (b) there exist no two threads such that both write to the same location (see §5 for details on generating the formula). The negation of such a formula is provided to a constraint-solver, like the SMT solver Z3. A similar check is done to ensure that all locations protected under locks are protected under a uniform lock amongst all threads. If the formula can be satisfied, there is a race allowed by the annotation. Otherwise, the co-ordination strategy expressed in the annotation ensures race-freedom, and if the program satisfies the annotation, it would be race-free as well.

For example, consider the outer-most loop at line 2 in Listing 1. Here we present the race freedom condition for two write accesses only. Two threads, corresponding to say indices i_1 and i_2 can access $C[i_1, \$y]$ and $C[i_2, \$y]$, provided $\$y$ satisfies the `where` clause. We hence form a constraint that asks whether there are two *distinct* i_1 and i_2 that may result in writing to the same position in C:

$$\exists i_1, i_2, \$y. \,(\, i_1 \neq i_2 \wedge \$y \geq 0 \wedge \$y < p \wedge i_1 = i_2 \wedge \$y = \$y \,)$$

It is evident that this formula cannot be satisfied. This is verified by feeding the formula to Z3. Hence, there is no race among the threads spawned at line 2, as long as the threads conform to the specified contract.

An example with a data race A natural question a programmer may wonder about is whether the middle loop in the algorithm in Listing 1 can be parallelized. If the *for* loop at Line 6 is changed to *foreach* loop, its annotation would look like: `reads` $A[i, j], B[j, \$y], C[i, \$y]$ `writes` $C[i, \$y]$. The annotation adequacy phase will however fail.

Transforming the program to test if it conforms to the co-ordination strategy (testing annotation correctness): While the verification that the co-ordination strategy ensures race-freedom gives some degree of assurance, we still do not know whether the program itself adheres to this strategy. In other words, we would like to be able to check whether the program satisfies its ACCORD specification, i.e. we want to verify whether the set of memory locations that may be accessed by a thread is contained in the set of memory locations specified in its annotation.

In order to answer this question, we propose to transform the formal ACCORD specification to *assertions* in the parallel program, so that checking these assertions at runtime during testing will in effect check whether the program meets the co-ordination strategy. ACCORD annotations of functions allow us to insert these assertions locally, checking whether the accesses at any point in the program are in accordance with the read and write regions specified by the local annotation in scope.

For example, we can transform the program in Listing 1 such that every access to a shared variable by a thread is preceded by a check that verifies that the access conforms to its ACCORD annotations. In general, this will require *storing* the values of variables at the fork-points, and inserting appropriate assertions in the code using these memoized variables. The parallel program with assertions corresponding to Listing 1 is given (on page 8) in Listing 6.

3. Parallel Programs

We define a simple parallel programming language, given by the following grammar, that resembles an imperative language like C, but with only boolean and integer types, and their multi-dimensional arrays, and with parallel-loop constructs and locks. The language also has the ACCORD annotation language built in.

```
⟨pgm⟩     ::= ⟨decl⟩* ⟨fun⟩*
⟨fun⟩     ::= ⟨type⟩ f(⟨decl⟩*)[requires ⟨φ⟩]⟨annot⟩*{⟨stmt⟩}
⟨stmt⟩    ::= ⟨decl⟩ | ⟨loc⟩ := ⟨expr⟩ |
                if ⟨bexpr⟩ then ⟨stmt⟩ else ⟨stmt⟩ |
                skip | return⟨expr⟩ | ⟨stmt⟩; ⟨stmt⟩ |
                while ⟨bexpr⟩ do {⟨stmt⟩} | ⟨parstmt⟩ |
                for(⟨type⟩i := ⟨expr⟩; ⟨bexpr⟩; ⟨stmt⟩){⟨stmt⟩} |
                synchronized (l){⟨stmt⟩}
⟨parstmt⟩ ::= thread⟨annot⟩*{⟨stmt⟩}
        | foreach(⟨type⟩i := ⟨expr⟩; ⟨bexpr⟩; ⟨stmt⟩)
                [requires ⟨φ⟩]⟨annot⟩*{⟨stmt⟩}
        | par [requires ⟨φ⟩]{⟨parstmt⟩ with ⟨parstmt⟩}
```

$$\begin{array}{lll}
\langle \mathit{loc} \rangle & ::= & i \mid i[\langle \mathit{aexpr} \rangle^*] \\
\langle \mathit{expr} \rangle & ::= & \langle \mathit{aexpr} \rangle \mid \langle \mathit{bexpr} \rangle \mid f(\langle \mathit{expr} \rangle^*)[\texttt{requires } \langle \phi \rangle] \\
\langle \mathit{bexpr} \rangle & ::= & \texttt{true} \mid \texttt{false} \mid \langle \mathit{aexpr} \rangle \langle \mathit{rop} \rangle \langle \mathit{aexpr} \rangle \mid \\
 & & \langle \mathit{bexpr} \rangle \texttt{ or } \langle \mathit{bexpr} \rangle \mid \langle \mathit{bexpr} \rangle \texttt{ and } \langle \mathit{bexpr} \rangle \mid \texttt{not } \langle \mathit{bexpr} \rangle \\
\langle \phi \rangle & ::= & \texttt{forall } i \texttt{ in } [\langle \mathit{aexpr} \rangle, \langle \mathit{aexpr} \rangle].\langle \phi \rangle \mid \\
 & & \texttt{exists } i \texttt{ in } [\langle \mathit{aexpr} \rangle, \langle \mathit{aexpr} \rangle].\langle \phi \rangle \mid \\
 & & \langle \phi \rangle \texttt{ and } \langle \phi \rangle \mid \langle \phi \rangle \texttt{ or } \langle \phi \rangle \mid \texttt{ not } \langle \phi \rangle \mid \langle \mathit{bexpr} \rangle \\
\langle \mathit{aexpr} \rangle & ::= & c \in \mathbb{N} \mid \langle \mathit{loc} \rangle \mid \langle \mathit{aexpr} \rangle \langle \mathit{aop} \rangle \langle \mathit{aexpr} \rangle \\
\langle \mathit{decl} \rangle & ::= & \langle \mathit{type} \rangle i \\
\langle \mathit{type} \rangle & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{lock} \mid \\
 & & \texttt{int}[\langle \mathit{aexpr} \rangle^*] \mid \texttt{bool}[\langle \mathit{aexpr} \rangle^*]
\end{array}$$

$$\begin{array}{lll}
\langle \mathbf{annot} \rangle & ::= & (\texttt{reads } \mid \texttt{ writes })\langle \mathit{region} \rangle \ [\texttt{under lock } l^*] \\
\langle \mathbf{region} \rangle & ::= & \langle \mathit{loc} \rangle \ [\texttt{where } \langle \phi \rangle]
\end{array}$$

i, f, l are identifiers
$\langle \mathit{aop} \rangle$: arithmetic operators, like $+$, $-$, $*$, $/$, % (modulo), etc.
$\langle \mathit{rop} \rangle$: relational operators on numbers, like $<$, $=$, etc.

A program has a sequence of global variable declarations followed by a list of functions. Each function has a return type, a name and a declaration of local variables followed by a sequence of statements, where statements include assignments, conditionals, sequential loops, synchronized blocks, or parallel statements. A parallel statement can be a `foreach` loop, which forks a separate thread for executing each different iteration of the loop. We assume `foreach` loops specify a lower bound, an upper bound, and a stride, which is similar to Fortran loops. All the threads spawned at this point must finish before the subsequent statement is executed. Hence `foreach` loops implicitly give fork-join parallelism. (Note that the runtime may actually schedule these parallel iterations on system threads in any manner; all that is assured is the causal dependence of the tasks and the fork- and join-points.) A parallel statement can also be defined using the `with` construct, which is a parallel composition operator that executes two statements in parallel. The language allows expressing nested fork-join parallelism.

The `synchronized` (l) block statement allows accesses the block to execute under a lock l. Note that this syntax allows *nested locking* only and furthermore, given any point in a function of the program, we can syntactically determine the set of locks that have been acquired since the beginning of the function.

The `foreach` construct can be optionally augmented with an *annotation* ($\langle \mathit{annot} \rangle$). Such a parallel-loop annotation declares, for each thread spawned at this point, the set of memory locations it will access (read and write). A write access is considered stronger than a read access; therefore if a location is both read and written to, it only needs be specified as a write annotation. An optional `where` clause specifies bounds for the variables in the `reads` and `writes` clauses. The `under lock` clause expresses that a memory location is accessed only when the prescribed lock is held by the executing thread. The annotation language also has a `requires` clause that allows making general assumptions on the state of the program variables when the forking happens. These are *pre-conditions* that the programmer asserts they know at the forking-point and are needed to argue why the co-ordination strategy is race-free (for example, this clause may say that the values in the array `A[]` are all distinct, in a program with parallelism using indirection arrays). The `where` and `requires` clauses are allowed to have *bounded quantifiers* to state properties about entire arrays, etc. However, we require that all auxiliary variables mentioned in the clause are quantified. The bounded range of the quantification is required to test the property at runtime.

Function declarations can also be annotated with read/write regions (the memory locations read and written by the function). For nested parallel blocks, we require that the outer parallel loop's annotation declares access to program variables (without holding any locks) that are used in inner blocks' annotation. Although the annotation language describes the general syntax, we abuse the notation a bit by grouping locations and regions in our examples (e.g. `reads` x, y stands for `reads` x `reads` y).

The annotation language has no *execution semantics*; it is solely designed to help the programmer annotate the precise parts of the memory accessed by each thread, along with the assumptions it makes, in order to argue that the strategy results entails race-freedom.

Let us assume that there is a `main` function, where the program starts, and that there are no calls to this function. Let us also assume that `foreach` loop indices (and other variables mentioned in the loop declaration) are never modified in the loop body. Notice that we do not support any form of aliasing in our language nor do we allow throwing exceptions. Handling aliasing in our framework can be achieved, and will basically proceed through an integrated static region-based alias analysis. However, in our current framework we require that parameters of a function do not alias. Exception handling can also be achieved in our framework by ensuring that an exception in one thread does not abort the execution of parallel threads. However, these will make the exposition of our thesis too complex, and we delegate this to future work.

Semantics The semantics of a program is the natural one: assignments, conditionals, loops etc. have the normal semantics; calls to functions are call-by-value. Assignments occur *atomically* even if they read multiple shared variables and write to a shared variable; it turns out that this is not unreasonable, as if we can prove a program race-free under these semantics, then the program with non-atomic assignments would also be race-free.

There is an implicit *barrier* at the end of the `foreach` loop and the `with` construct. The semantics ensures that all threads complete before the next statement is executed. The semantics of synchronized blocks is that the specified lock is acquired and released at the start and end of the block respectively.

An execution is hence a partial order of events that respects the above rules for the `foreach` and `with` constructs (i.e. events across the fork or across a barrier are ordered in the right way) and the locking mechanism, and furthermore is *sequentially consistent* (i.e. events in one thread must respect the program order).

Data Race Two operations in an execution that are *not* acquisitions or releases of locks are said to be in *conflict* if they access the same memory location and at least one of them is a write. A program has a *data-race* if there is some *sequentially consistent* execution which has two operations in conflict that are not ordered (in other words, there is a sequentially consistent execution after which two conflicting operations are enabled). A program is *data-race free* if it has no data-races.

The above definition of data-races using an ideal sequentially consistent semantics is the standard one (see [2] for instance). Note that races on locks are not considered data-races. Programming languages such as Java and C++ have similar semantics and definition of data-races; furthermore, the memory model of these languages assures us that if the program is data-race free (according to the sequentially-consistent memory model as above), the actual memory model will ensure sequential consistency. This seemingly circular definition (data-races are defined using the sequentially consistent model and lack of data-races ensures the weak memory model assures sequential consistency) is standard and known as the *data-race freedom guarantee* (DRF guarantee). Our goal is to write programs with annotations that we can prove data-race free, and hence assure sequential-consistency in any memory model that has a DRF-guarantee.

4. Accord Annotations for Parallel Programs

In this section, we present several examples of data-parallel programs and annotate them using Accord contracts. The programs illustrate the expressiveness and succinctness of our annotation language. The suite includes a modified version of the successive over-relaxation (SOR) algorithm, a fully parallel quicksort algorithm (with parallel partitioning and sorting) [21], a parallel implementation of the MonteCarlo simulation program (only the parallel component) from the Java Grande benchmark suite [27], and a sparse matrix multiplication routine with parallelism using indirection arrays. Note that the programs have been annotated by the authors.

4.1 Successive Over-Relaxation with Red-Black Ordering

Successive over-relaxation (SOR) is a variant of the Gauss-Siedel method for solving a linear system of equations, which results in faster convergence. The elements in the equation matrix can be reordered in such a way that alternate elements are marked as red and black (hence the name red-black ordering), giving a checker board pattern. Importantly, in each iteration of SOR all the red elements can be updated in parallel while reading the black elements, followed by a barrier and then an update of the black elements while reading the red elements, thus avoiding races.

Listing 2 shows a parallel implementation of the routine that is executed in each iteration of the SOR algorithm. The program takes a matrix $A[m, n]$ as input, and updates the elements in the matrix A. This code expresses very fine-grained parallelism since all elements of one kind (red or black) are updated in parallel.

Listing 2 Successive Over-Relaxation with Red-Black Ordering

```
1  void sor (int[m,n] A, int w) {
2
3   //update red
4   foreach (int id := 1; id < m; id:=id+1)
5    reads w, A[id,$j],A[id-1,$j],A[id+1,$j],
6      A[id,$j-1],A[id,$j+1] writes A[id,$j]
7    where 0 <= $j and $j < n and ((id+$j) % 2 = 0){
8
9     foreach(int k := 2-(id%2); k < n; k := k+2){
10     reads w, A[id,k], A[id-1,k], A[id+1,k],
11        A[id,k-1], A[id,k+1]
12     writes A[id,k]
13     requires ((id + k) % 2 = 0) {
14
15      A[id,k] := (1 - w) * A[id,k] + w * 0.25 *
16        (A[id-1,k]+A[id+1,k]+A[id,k-1]+A[id,k+1]);
17    } }
18
19  //update black
20  ...
21 } } }
```

The `foreach` loop at line 4 divides the matrix among threads in a row-wise fashion. In each iteration of this loop, the red elements in a given row are read and written, while the adjacent black elements are read only. In a checker board pattern, a simple way to check the membership of each element is to test whether the sum of its indices (x, y) is odd or even, which can be expressed using arithmetic constraints (an element (x, y) is red iff $(x+y)\%2 = 0$). As Listing 2 suggests (lines 5-7), Accord annotations can express such a sharing strategy by allowing complex modulo arithmetic constraints in the `where` clause. Also note that the annotation uses an *auxiliary* variable $\$j$, which is free, to specify the columns that will be accessed by a thread.

The inner `foreach` loop (line 9), which further divides the elements in a row among threads, is annotated similarly. The only difference is that the natural way to write this annotation is to use the two loop variables to specify the memory locations read and written by each thread (id and k). The second outer loop (beginning at line 20 and not shown here) is similar to the first loop except that it reads both red and black elements, and updates black elements only.

4.2 Quicksort

We implement a race-free, fully parallel algorithm for quicksort [21]. While many parallel implementations of quicksort only exploit the inherent divide-and-conquer parallelism, our implementation performs the partitioning of the array around a pivot in parallel as well, using a parallel rank (prefix sum) algorithm.

Listing 3 shows our implementation. The main function takes an array $A[n]$ and sorts its contents by recursively calling itself. Due to limited space, we have not shown the functions *rel_pos_rec* and *write_pos_rec*, which exhibit a similar recursive parallelism.

Listing 3 Quicksort algorithm with parallel partitioning.

```
1  void qsort (int[n] A, int i, int j) reads i, j
2     writes A[$k] where (i <= $k and $k < j) {
3    if (j-i < 2) return;
4    int pivot := A[i];  //first element
5    int p_index := dyn_partition(A, pivot, i, j);
6    // swap 1st element and element at p_index
7
8    par requires p_index >= i and p_index < j {
9    thread writes A[$k] where (i<=$k and $k<p_index)
10       { qsort(A, i, p_index); }
11   with
12   thread writes A[$k] where (p_index<$k and $k<j)
13       { qsort(A, p_index + 1, j); }
14   }
15 }
16
17 int dyn_partition(int[n] A,int pivot,int i,int j)
18     reads i, j, pivot
19     writes A[$k] where (i<=$k and $k< j) {
20
21   int[n] temp, B;
22   int smalls := rel_pos_rec(A,temp,pivot,i,j);
23   write_pos_rec(A,B,temp,pivot,i,j,0,smalls);
24   A := B;
25   return smalls;
26 }
```

In addition to being a complex program, it highlights some interesting features of our annotation language. The `thread` statement can be annotated (lines 9 and 12), similarly to the `foreach` statement, and the annotation specifies the set of memory locations read and written by the thread. This example also requires annotating functions (lines 2 and 18-19). Function annotations, which include `reads` and `writes` clauses, are important as they give local read and write regions that can be used for modularly testing the program against the annotations. The quicksort program also shows that recursive partitioning can be handled by our annotation language in addition to iterative partitioning of data.

4.3 MonteCarlo Simulation

MonteCarlo is a multi-threaded benchmark from the Java Grande suite. It uses Monte Carlo techniques to find the price of a product based on the price of an underlying asset. The given code sequentially generates N tasks, each with a different parameter. During the parallel phase, these tasks are divided among a group of threads in a block fashion. At the end of processing each task, the corresponding thread writes the simulation result back into a list of results, which is *shared among the threads*. The accesses are protected by a lock. After the parallel phase, the results are reduced in a sequential fashion.

Listing 4 Simplified version of the MonteCarlo simulation code from Java Grande.

```
1  void main(int[nTasks] tasks) {
2    int slice := (nTasks+nThreads-1)/nThreads;
3
4    foreach (int i:=0;i<nThreads;i:=i+1)
5      reads next under lock gl, tasks[$k]
6      writes next, results[$j] under lock gl
7      where (i*slice)<= $k and $k<((i+1)*slice)
8      and 0 <= $j and $j < nTasks {
9
10     int ilow := i*slice;
11     int iupper := (i+1)*slice;
12     if (i = nThreads-1) iupper := nTasks;
13
14     for(int run:=ilow; run<iupper; run:=run+1){
15       int result := simulate(tasks[run]);
16
17       synchronized (gl) {
18         next := next + 1;
19         results[next] := result;
20       }
21 } } }
```

Listing 4 shows a simplified version of the program for illustration. Some functions have not been shown due to limited space. The main function takes an array $tasks[n]$ as input and processes each element within the array using parallel threads. Note the program acquires a lock at line 16 before writing to the shared variables $next$ and $results$, and then releases the lock implicitly at line 19.

4.4 Sparse Matrix Multiplication

In Listing 5, we show the simplified core of the Sparse Matrix Multiplication program from the Java Grande suite. The annotation on lines 7-8 specifies that each thread writes to $yt[row[\$j]]$, hence the answer to whether the loop is race-free depends on the runtime contents of the indirection array $row[]$. The subsequent pre-condition (lines 10-14) specifies a property of this array which helps in firmly establishing the race-freedom of the loop. We conjecture that such a complex strategy is in the programmer's mind during the design or implementation of the algorithm, but is hard to statically infer automatically. By declaring it using ACCORD's annotation language makes it easier to reason about the race-freedom of the program.

5. Annotation Adequacy: Generating Non-interference Conditions

The annotation adequacy phase in ACCORD checks whether the contract implies race-freedom, i.e. whether *any* program that satisfies the contract is race-free. We construct a verification condition from the annotation which is then checked by Z3, an SMT solver from Microsoft Research. The verification condition is constructed such that it is satisfiable if the annotations are insufficient to prevent a memory race at some particular memory location. Hence, if the condition is satisfied, then a satisfying solution gives *valuable debugging information*: two threads and a memory location involved, and a program state sufficient for both thread annotations to grant simultaneous access to the location. On the other hand, if the conditions are unsatisfiable then no two concurrent threads that satisfy the annotation can race.

The verification condition is constructed in two parts. One part checks the regions written to by one thread are disjoint from the regions read or written by any other thread created in the same parallel construct. Thus no data race involves sibling threads. The next part checks that the regions declared for a function call or parallel statement are subsets of the regions declared for the surrounding thread. By induction up to a common ancestor, this ensures no two threads which might execute concurrently can have a data race.

Listing 5 Simplified version of the Sparse Matrix Multiplication code from Java Grande.

```
1  void sparsematmult(int nThreads,
2                 int[] yt, int[] x, int[] val,
3                 int[m] row, int[n] col,
4                 int[] lowsum, int[] highsum) {
5    foreach (int id = 0; id < nThreads; id:=id+1)
6
7     reads x[col[$j]], val[$j] writes yt[row[$j]]
8      where $j >= lowsum[id] and $j < highsum[id]
9
10    requires (forall $t1 in [0,nThreads-1],
11      $t2 in [0,nThreads-1]. ($t1 != $t2 implies
12      (forall $x1 in [lowsum[$t1],highsum[$t1]-1],
13        $x2 in [lowsum[$t2],highsum[$t2]-1].
14           row[$x1] != row[$x2] ))) {
15
16    for (int i=lowsum[id]; i<highsum[id]; i:=i+1){
17          yt[ row[i] ] += x[ col[i] ] * val[i];
18 } } }
```

5.1 Non-interference with a parallel statement

Consider a parallel loop of the form: `foreach`(int i:=1, cond(i), i:=i+1) with a requires formula ψ, a set of read annotations R and a set of write annotations W. Given an annotation a in one of these sets, let $arr(a)$ be the array mentioned in the annotation (scalar variables are treated as 0-dimensional arrays), and $locks(a)$ be the set of locks on the annotation (which may be empty). The annotation is parameterized over the loop index i; $a(i)$ is the formula describing the region for iteration i. The array indices are formal parameters of this formula. It may include program variables, however, any other variables must be quantified.

We treat `par` blocks similarly to a `foreach` loop with the index ranging from 1 to 2, with appropriately adapted annotations.

Let $Conforms(A, \vec{x}, \varphi)$, abbreviated $[\![A[\vec{x}] \in \varphi]\!]$, be the formula that expresses that access to the location $A[\vec{x}]$ is allowed by the annotation φ. The formula is simply *false* if the A is not the array mentioned in the annotation, otherwise the formula is formed by replacing the formal parameters in the boolean expression in φ with the corresponding expressions in $\vec{x}$.

For instance, consider the `writes` annotation for C on line 3 in Listing 1. Then, for a parallel iteration with loop-index i_1, $[\![C[i,k] \in (C[x,y] \texttt{ where } 0 \leq x \wedge x < n \wedge 0 \leq y \wedge y < p)]\!]$ $= (0 \leq i \wedge i < n \wedge 0 \leq k \wedge k < p)$.

We are now ready to define the constraint that is satisfiable iff the annotation on a single parallel `foreach` construct allows a race between child threads. Consider a `foreach` loop with read annotations R, write annotations W, and a requires clause ψ. Then we generate the following constraint:

$$\psi \wedge \exists i, j, \vec{x}.\ [\ (i \neq j) \wedge$$

$$\bigvee_{A \in arr(R) \cup arr(W)} \left(\bigvee_{\substack{w \in W, a \in R \cup W \\ arr(w) = arr(a) \\ locks(w) \cap locks(a) = \emptyset}} [\![A[\vec{x}] \in w(i)]\!] \wedge [\![A[\vec{x}] \in a(j)]\!] \right)]$$

The formula asserts that there is a single location which two distinct iterations i and j either both write into, or one of them writes to and the other reads from, without holding some common locks. For each parallel block, the above formula is constructed, translated into the SMT-LIB[23] syntax accepted by Z3, and then checked. This formula is satisfiable iff the annotation permits races.

5.2 Checking inclusion of nested annotations

The previous section describes the check that the regions described in the annotations for each thread created by a parallel statement are disjoint. However, we also want the annotations at various levels in a program be *consistent* with each other. Apart from ensuring the sanity of the annotations, this check is important for the testing phase, as we can simply check whether the program at any point reads and writes to the region specified in the annotation immediately surrounding the access. This check is done by writing a formula which can be satisfied only if there is some location which may be accessed under the child annotations but not the parent's annotation.

If the child statement is a function call the function annotation will be written in terms of the formal parameters of the function. The actual parameters in the call statement need to be substituted for the actual parameters before checking inclusion. Also, function annotations may give the surrounding context if we are checking a child statement which is not lexically within any parallel construct.

One complication is that a program variable may be mentioned in both sets of annotations, but have different values at the two different points. This can be handled by substituting fresh variables for free variables corresponding to program variables that may be modified. In particular, we prohibit assignments to the index of a `foreach` loop, so these variables may be shared. To handle locks, we find the set L of additional locks held at the child statement, and refer to it while constructing the verification condition.

The check will be described for the most general case, which is a pair of nested `foreach` loops. A function annotation can be treated as if it came from a loop with a single iteration, whose annotation does not mention the loop index.

Inclusion fails if access to some location is granted by one of the child's annotations, but not by any of the parent's annotations. Let R_p and W_p be the read and write annotations for the parent, R_c and W_c be the read and write annotations for the child, and ψ_c and ψ_p be the requires formula for the child and parent. Note the child annotations may be parameterized over an inner and an outer thread index. The formula for checking read annotations is

$$\exists i, j, \vec{x}.\,[\;\psi_c \wedge \psi_p \wedge$$
$$\bigvee_{r \in R_c} [\![arr(r)[\vec{x}] \in r(i,j)]\!] \wedge \neg\Bigg(\bigvee_{\substack{p \in R_p \cup W_p \\ arr(p) = arr(r) \\ locks(p) \subseteq L \cup locks(r)}} [\![arr(r)[\vec{x}] \in p(i)]\!] \Bigg)]$$

and for write annotations

$$\exists i, j, \vec{x}.\,[\;\psi_c \wedge \psi_p \wedge$$
$$\bigvee_{w \in W_c} [\![arr(w)[\vec{x}] \in w(i,j)]\!] \wedge \neg\Bigg(\bigvee_{\substack{p \in W_p \\ arr(p) = arr(w) \\ locks(p) \subseteq L \cup locks(w)}} [\![arr(w)[\vec{x}] \in p(i)]\!] \Bigg)]$$

If the parent annotations contain no existentially quantified auxiliary variables, this formula contains no nested quantifiers. Otherwise, negating membership in the parent regions produces universal quantifiers. This can be avoided if auxiliary variables are only used on annotations for function or parallel statements which contain no function calls or parallel statements. In our experiments, all inclusion checks resulted in formulas with only existential quantification, and can thus be handled reliably by an SMT solver.

6. Transforming Programs for Testing Against Annotations

While the previous phase ensures that the co-ordination strategy ensures race-freedom, it does not ensure that the program itself implements the strategy correctly. The objective of this section is to describe how to transform a parallel program with ACCORD annotations to one with assertions, such that on any test run, the assertions check whether the program correctly implements the co-ordination strategy.

Note that testing tools can, of course, check whether executions have data-races or not. However, the testing that we are achieving here is more general— we are checking whether the executions meet the *co-ordination strategy*. Therefore, even if a certain run does not exhibit data-races, the run could still violate the declared co-ordination strategy, and thus hint at the presence of data-races in other executions in the program.

For example, consider a forking point that creates two threads T_1 and T_2, and consider an execution where T_1 writes to x, followed by a lock-protected access to y, and then T_2 does a lock-protected access to y followed by a write to x. In this execution, there is no data-race, as the lock-protected accesses to y make the accesses to x causally ordered (and an accurate testing tool that keeps track of the happens-before relation will not detect the race). However, there is no way to annotate the threads in any way such that the annotation implies race-freedom and the execution satisfies the annotation (indeed, such an annotation cannot allow T_1 and T_2 to both write to x without being under a common lock).

The transformation of the parallel program to one with assertions is done by specializing the annotation to each thread and translating them to runtime assertions which are then inserted in the thread body.

We do not give a formal description of how the assertions are inserted, but go through the main components and insertion procedures. First, note that in a function, we need to check whether the accesses in the function correspond to only the annotation of the function, and need not check whether the annotation of a caller of the function is also satisfied. This is primarily because of the fact that our inclusion checks ensure that the annotations of the function define subregions of the regions defined by any caller. Similarly, inside nested parallel loops, we need to check accesses against only the annotations at the innermost parallel loop.

Next, in order to check the `requires` clauses in the annotation, note that they allow only *bounded* quantification over ranges, and hence can be checked using a series of nested loops that range over the domains. We can then insert assertions that check if the specified pre-condition holds.

Finally, turning to the accesses, one complication that arises is that an access to an array $A[\vec{x}]$ may be within an annotation a, but the program variables used in a may have changed before the access. We hence insert, at the point of the annotation, assignments to new variables that *cache* the value of the program variables at the annotation site. These variables never get modified, and are used at the access points to check if the indices $\vec{x}$ satisfy the conditions demanded by the annotations. More precisely, for every access to $A[\vec{x}]$, we first figure out statically the set L of locks acquired since the beginning of the `foreach` loop or function immediately surrounding this access. If the access is a read access we insert an assertion that checks the following formula, where R and W are the enclosing set of read and write annotations, and i is the index of the enclosing loop (replace $p(i)$ with p if the access is not surrounded by a loop).

$$\bigvee_{\substack{p \in R \cup W \\ arr(p) = A, locks(p) \subseteq L}} \qquad [\![A[\vec{x}] \in p(i)]\!]$$

Table 1. Summary of results from evaluating ACCORD.

	Lines of code	# Parallel Loops/Function Annotation	*Annotations* # `reads/` `writes` clauses	# `where` clauses	# `lock` clauses	# Pre cond.	*Adequacy phase* Logic used	Time taken	Success? (Yes/No)	Proven Race Free?
MatMult	25	2	2/2	1	0	0	QF_LIA	<1s	Yes	Yes
MatMult (buggy)	30	3	3/3	1	0	0	QF_LIA	<1s	No	No
SOR	45	4	4/4	4	0	2	QF_NIA	<1s	Yes	Yes
Quicksort	100	-	4/7	9	0	0	QF_LIA	<1s	Yes	Yes
LuFact	35	1	1/1	1	0	0	QF_LIA	<1s	Yes	Yes
LuFact (buggy)	35	1	1/1	1	0	0	QF_LIA	<1s	No	No
`montecarlo-jgf`	255	1	1/1	1	1	0	QF_UFLIA+MA	<1s	Yes	Yes
`sparsematmult-jgf`	50	1	1/1	1	0	1	AUFLIA	<1s	Yes	Yes
`series-jgf`	800	1	1/1	1	0	0	QF_UFLIA+MA	<1s	Yes	Yes
`moldyn-jgf`	1300	6	5/6	6	0	0	QF_LIA	<1s	Yes	Yes
`wupwise_l`	1029	16	14/14	0	1	4	QF_NIA	<1s	Yes	Yes
`swim_l`	275	12	12/12	0	0	0	QF_LIA	<1s	Yes	Yes
`mgrid_l`	722	22	21/20	0	0	0	QF_LIA	<1s	Yes	Yes
`applu_l`	2586	33/4	30/34	4	0	0	QF_NIA	<1s	No	No (8)
`gafort_l`	691	9/4	12/13	0	1	1	QF_LIA	<1s	Yes	Yes
`art_l`	1594	5/7	12/11	1	1	0	QF_LIA	<1s	Yes	Yes

QF_LIA - Quantifier Free Linear Integer Arithmetic, **QF_NIA** - Quantifier Free Non-Linear Integer Arithmetic, **AUFLIA** - Arrays and Linear Integer Arithmetic with Uninterpreted Functions, **QF_UFLIA+MA** - Quantifier Free Linear Integer Arithmetic with Uninterpreted Functions and Multiplication Axioms

In this expression, the program variables are replaced by the cached program variables to build the assertion (and bounded quantification in `where` clauses are converted to loops). Similarly, for a write access to $A[\vec{x}]$, insert an assertion that checks the following formula.

$$\bigvee_{\substack{p \in W \\ arr(p)=A, locks(p) \subseteq L}} [\![A[\vec{x}] \in p(i)]\!]$$

Consider the matrix multiplication program given in Listing 1. The transformed program with assertions is given in Listing 6. The first three assertions (lines 7-9) correspond to the first annotation and the accesses to A, B and C, while the last three assertions (lines 11-13) correspond to the second annotation.

Listing 6 Transformed parallel program with assertions corresponding to the parallel implementation of matrix multiplication in Listing 1

```
1  void mm (int[m,n] A, int[n,p] B) {
2    foreach (int i := 0; i < m; i:=i+1) {
3      n':=n; p':=p;
4      for (int j := 0; j < n ; j:=j+1)
5        foreach (int k := 0; k < p; k:=k+1) {
6          assert(i=i and 0<=j and j<n');
7          assert(0<=j and j<n' and 0<=k and k<p');
8          assert(i=i and 0<=k and k<p');
9
10         assert(i=i and j=j);
11         assert(j=j and k=k);
12         assert(i=i and k=k);
13
14         C[i,k] := C[i,k] + (A[i,j] * B[j,k]);
15 } }    }
```

Races in the transformed program: The transformed program P' could have data-races. However, we argue that this does not interfere with checking whether the original program P satisfies its annotations.

First, let us assume that in any annotation, the variables mentioned in the annotation are included in the read-set defined by the immediately surrounding annotation. Let us also assume that the annotation adequacy phase passes, and hence we know that if P satisfies its annotations, then it is race-free.

Next, note that P satisfies its annotations iff P' satisfies its assertions on all sequentially consistent runs. This follows from the fact that we have introduced assertions precisely to check the annotations.

We now want to argue that P satisfies its annotation iff P' satisfies its assertions in any run on a memory model with a DRF guarantee.

First, we argue that if P satisfies its annotations, then P' will be race-free. This is easy to see; P' reads from variables mentioned in the annotation which are already covered by the surrounding read annotation, and hence P' accesses only variables that are allowed by the annotation of P, which ensures race-freedom. Hence P' is race-free, and hence we can run P' on any memory model with a DRF guarantee, which will result only in sequentially consistent runs, and hence satisfy all its assertions.

Conversely, if a run of P' (on a DRF memory model) fails an assertion, then either the run is sequentially consistent or not. In the former case, we know that P also does not meet its annotation on a sequentially consistent run. In the latter case, by the DRF guarantee P' must not be race-free, and hence by the argument above P cannot satisfy its annotations.

Hence, we can test whether P satisfies its annotations by testing P' on any memory model that has a DRF guarantee.

7. Evaluation

In this section, we describe our experience in augmenting parallel programs with ACCORD in order to check race-freedom. We provide statistics of annotation burden and show that the ACCORD annotations are minimal and do not put an undue burden on the programmer, and that the annotation language can express the sharing in complex and realistic programs. We evaluated over three suites of benchmarks, which we describe below.

Suite 1: Apart from the examples that we have described in sections §2 and §4 (matrix multiplication, SOR, and QuickSort), we also annotated and checked parallel LU Factorization (LuFact).

Suite 2: This includes four programs from the Java Grande Forum (JGF) benchmark suite: `montecarlo` (MonteCarlo simulation), `series` (which computes Fourier coefficients of a function), `moldyn` (an N-body code modeling interacting particles) and `sparsematmult`. These programs are parameterized by the number of threads: during execution, the threads divide the data among themselves by computing a non-linear formula over the total num-

ber of threads and the size of the data. This formula forms the region of computation for each thread.
Suite 3: The last suite of benchmarks is part of the Spec OMP2001 Suite (v3.2), consisting of fairly large programs written in OPENMP. It was significantly harder to understand and annotate the co-ordination strategy in these programs, though the number of annotations required were only a small fraction of the code.

We were able to annotate all these programs and check them for race-freedom using ACCORD. Table 1 presents the results of our evaluation. The name and the code size of our benchmarks are given in the first two columns. Some benchmarks also require annotating functions that are called from parallel blocks for modular reasoning. Note that these are considerably larger programs (up to 2586 LOC and up to 33 fork-join loops).

Annotations The next five columns (labeled *Annotations*) of Table 1 provide information about the annotations required for these programs. The fourth column records the number of total `reads` and total `writes` clauses required to annotate each program. The number of `where` and `lock` clauses in each program are listed in the next column. The last column under *Annotations* lists the number of pre-conditions in the program.

The annotation statistics show that the additional burden on the programmer incurred in writing ACCORD annotations is not much, especially in the larger programs. These results and our experience suggest that the annotations themselves are a natural way to express the parallel co-ordination strategy. We tested the annotations for several of these programs by transforming the parallel program to include assertions (as described in Section 6), and by executing them on test inputs. We believe ACCORD annotations can simplify both manual and automatic reasoning significantly.

Checking Annotation Adequacy: As discussed earlier, we generate a verification condition which is fed to Z3 in order to prove that the annotations imply race-freedom. The columns under the adequacy phase in Table 1 show the results of checking constraints generated for non-interference between sibling threads. Note that we did not find nested parallel blocks in any of the programs.

The first column in the adequacy phase gives the logic used to prove whether the verification condition is satisfiable. We use the SMT_LIB notation [23]. Note that most programs are proved race-free using linear integer constraints. Proving the SOR, `wupwise_l` and `applu_l` program requires non-linear integer arithmetic. The verification condition for certain benchmarks includes multiplication and division, which Z3 is unable to handle. Therefore, we use a linear integer arithmetic with uninterpreted functions (UF), and model multiplication as an uninterpreted function with basic axioms that capture its properties. This allowed us to prove the race-freedom property of MonteCarlo, `series` and `sparsematmult`.

The next column reports the time taken by Z3 per parallel loop (the time taken is minimal), and whether the annotations implied race-freedom. The last column reports whether ACCORD could prove the program to be race-free. For both buggy programs, the annotation adequacy check failed i.e., Z3 is able to prove the existence of a data-race, given the verification condition generated from the program annotations. While we intentionally introduced a race in the matrix multiplication algorithm (see §2), the data-race in the LU Factorization was an unintended bug in our implementation. The data-race occurs due to an overlap between a read set and a write set (of different threads) due to a subtle boundary condition.

Finally, and surprisingly, we also discovered a set of (eight) previously unknown data-races in the `applu_l` benchmark from the Spec OMP2001 suite, which has been available as a benchmark for more than 10 years. The bugs have been reported and duly acknowledged. Seven of these data-races have a similar bug pattern, which is triggered only under certain interleavings (the bug is caused because of the removal of a barrier between two for-each loops using a `nowait` clause). This further strengthens our claim that annotating programs with ACCORD can help discover bugs due to complex sharing strategies.

8. Related Work

There is a rich literature on using type analysis in order to ensure race-freedom [3, 4, 6, 8, 12, 13, 20]. Ownership types that statically enforce object-level encapsulation, combined with *effect* systems that capture computational effects, have been used to define nested regions, separate them, and ensure race-freedom. These systems have been extended for locks to statically ensure deadlock-freedom. The Deterministic Parallel Java language [6] combines types and effects to give the user the ability to give distinct names for regions, including nested regions, specify read and write effects on regions by parallel threads, and by ensuring disjointedness of regions, ensure race-freedom and even determinacy.

The main difference between our work and that of type systems is that our annotations allow *dynamic* and *complex logical partitioning* of the heap into different regions. It would be very hard to type check, for example, the Successive Over-Relaxation (SOR) example. The regions in SOR are *dynamic* (due to the phases) and even within a phase, the regions are not nested and are instead specified using logical constraints ($row + col$ is even/odd). Realizing such a program using static types or type annotations, even with dynamic ownership, is quite challenging, unless the program is rewritten using different data-structures or using copying data between different structures that have different regions. The complexity of course comes at a price— our analyses tackle an undecidable problem (while usually type-analysis is often decidable and fast), and we trade this in order to be able to express complex separation constraints. We instead rely on the emerging class of software verification and SMT solver technology to be effective in practice.

SharC [3] is a type system that assigns different kinds of sharing modes to objects, and these are enforced using a combination of static and dynamic techniques (dynamic techniques kick in when the static analyses fail). The work in [14] proposes contracts which allow fractional permissions, which can be verified using an SMT solver. These approaches deal with objects that are shared among different threads during their lifetime, and employ mutual exclusion synchronization primitives such as locks for correct behavior. In this paper, we need annotations for sharing complex sub-regions that are logically defined and dynamically evolve, and hence the mechanisms proposed in the above work do not suffice. However, combining our annotations with the annotations above to handle static simple region separation and locks would be interesting.

There is some recent work [15] that proposes *inferring* the read and write regions from loop-free (SPMD) CUDA [1] programs, and using SMT solvers to check whether these regions do not intersect. Note that CUDA programs are recursion free and function pointer free. We believe that such an inference may be very hard for larger programs with complex control and data structures. We have instead proposed annotation mechanisms that the user can write for a fairly general purpose programming language.

Separation logic [24] is a Hoare logic for reasoning about heap structures, especially separation, and hence is very relevant as a means of annotation to separate threads. Separation logic has been primarily used to separate dynamic heaps using recursion. When moving to programs with dynamic data, we certainly envisage using logics for separation.

There is a rich literature on *checking* concurrent programs for data-races, a posteriori, with no extra user annotations, using static analysis, testing and model-checking: lock-set based algorithms as in Eraser [25], vector-clock based algorithms [26], hybrid algorithms [22, 29], Goldilocks [11], and static-analysis algorithms [9].

Data-race-freedom or the similar property of non-interference has also been the focus of parallel compilers community under the broader problem of dependency analysis for loops and operations over arrays [28]. The primary motivation in their work is to automatically extract parallelism. We believe manual annotations go a long way in simplifying the problem (for instance, kernels like sparse matrix). In addition, we use sophisticated theorem provers to prove race-freedom. However, we believe that techniques from the auto-parallelization community can help in inferring many simple ACCORD annotations.

9. Discussion and Future Work

Race-freedom is a generic correctness condition for concurrent programs and, given that languages like the next version of C++ consider programs with races erroneous and do not even offer semantics for them, there is an urgent need for software engineering techniques to ensure programs are race-free.

We have, in this paper, proposed a mechanism by which a programmer can formally express the intended concurrency co-ordination strategy, for array-based programs with fork-join parallelism working over complex read/write regions. Expressing the co-ordination strategy formally allows us to check automatically if the strategy ensures race-freedom and also allows automatic insertion of assertions to test if the program conforms to the strategy. We believe that the ACCORD thread contracts and the accompanying reasoning mechanisms presented in this paper are elegant in capturing *complex* region separation for data-parallel programs to prove race-freedom.

The ACCORD annotations programmers write can have benefits other then ensuring race-freedom. There is a promising line of work that suggests that the annotations can make programs run faster—for instance, information about the separation of data can be made available through the compiler to the underlying architecture in order to provide efficient runtime execution mechanisms. The new architecture framework DeNovo being developed at Illinois aims to utilize precisely this kind of separation information and assurance of race-freedom to build simple cache-coherence and faster runtime architectures [19].

The ACCORD language proposed here is a first step towards building a language that can express concurrency co-ordination strategies. ACCORD has currently several limitations— the language cannot handle read/write regions of dynamically allocated heaps, and cannot handle non-fork-join parallelism. These are not inherent limitations to our approach; for example, there are simple mechanisms, such as the inductively-defined nested regions in DPJ [6], or separation logic annotations that can express regions of the dynamic heap. In fact, in ongoing work, we are pursuing an extension of the work presented in this paper with researchers working on DPJ [6], to design a more general and full-fledged annotation language, with associated static-checking and testing mechanisms.

References

[1] NVIDIA CUDA programming guide version 3.0. *http://www.nvidia.com/cuda*, 2010.

[2] S. Adve, M. Hill, B. Miller, and R. Netzer. Detecting data races on weak memory systems. In *The 18th Annual International Symposium on Computer Architecture, 1991.*, pages 234–243, 1991.

[3] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: checking data sharing strategies for multithreaded c. In *PLDI '08*, pages 149–158. ACM New York, NY, USA, 2008.

[4] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of java without data races. In *OOPSLA '00*. ACM Press, 2000.

[5] P. Becker. Working draft, standard for programming language C++. Technical report, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, 2010.

[6] R. Bocchino and V. A. et al. A type and effect system for deterministic parallel java. In *OOPSLA '09*. ACM, 2009.

[7] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *PLDI '08*, New York, NY, USA, 2008. ACM.

[8] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *OOPSLA '01*, pages 56–69. ACM, 2001.

[9] 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 *PLDI '02*, pages 258–269, 2002.

[10] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963/2008 of *LNCS*, pages 337–340. Springer Berlin, April 2008.

[11] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In J. Ferrante and K. S. McKinley, editors, *PLDI '07*, pages 245–255. ACM, 2007.

[12] C. Flanagan and M. Abadi. Object types against races. In *CONCUR*, volume 1664 of *LNCS*, pages 288–303. Springer, 1999.

[13] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI '00*, pages 219–232, New York, NY, USA, 2000. ACM.

[14] K. R. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP '09*, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] R. Lublinerman and S. Tripakis. Checking equivalence of spmd programs using non-interference. Technical Report UCB/EECS-2009-42, EECS Department, University of California, Berkeley, Mar 2009.

[16] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *POPL '05*, pages 378–391. ACM New York, NY, USA, 2005.

[17] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.161279.

[18] Microsoft Corporation. Code Contracts. *http://research.microsoft.com/en-us/projects/contracts/*, 2008-10.

[19] Parallel@Illinois. Denovo: Rethinking hardware for disciplined parallelism. *http://rsim.cs.illinois.edu/denovo/*, 2008-10.

[20] P. Permandla, M. Roberson, and C. Boyapati. A type system for preventing data races and deadlocks in the java virtual machine language: 1. In *LCTES '07*, page 10, New York, NY, USA, 2007. ACM.

[21] D. Powers. Parallelized Quicksort and Radixsort with Optimal Speedup. In *Proceedings of the International Conference on Parallel Computing Technologies, 1991.*, pages 167–176, 1991.

[22] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03*, pages 179–190, New York, NY, USA, 2003. ACM.

[23] S. Ranise and C. Tinelli. The smt-lib standard: Version 1.2. 2006.

[24] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings.*, pages 55–74, 2002.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 1997.

[26] D. Schonberg. On-the-fly detection of access anomalies. In *PLDI '89*, pages 285–297. ACM New York, NY, USA, 1989.

[27] L. A. Smith and J. M. Bull. A multithreaded java grande benchmark suite. In *In Proceedings of the Third Workshop on Java for High Performance Computing*, pages 97–105, 2001.

[28] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0805327304.

[29] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.