Thread Contracts for Race-Freedom

Rajesh K. Karmani P. Madhusudan Brandon M. Moore
University of Illinois at Urbana-Champaign
{rkumar8, madhu, bmmoore}@illinois.edu

***Abstract*—The goal of this paper is to build an annotation framework of thread contracts, called ACCORD to argue that a parallel program has no data-races, and build accompanying verification and testing tools. ACCORD annotations allow programmers to declaratively specify the fine-grained parts of memory that a thread may read or write into, and the locks that protect them, and hence can be used to establish race-freedom. We show that this can be achieved using automatic *constraint-solvers* based on SMT-solvers. We also show how to compile ACCORD thread contracts to *runtime* assertions that check the contracts dynamically during testing. Furthermore, we explore static verification of annotation correctness for parallel programs, using a new and surprising reduction to verifying assertions in *sequential programs*; the latter can be tackled using sequential contract-verification tools. Using a large class of data-parallel programs that share memory in intricate ways, we show that natural and simple contracts suffice to argue race-freedom, and that the task of showing that the annotations imply race-freedom and the task of showing the annotations are respected by the program, can be handled by existing SMT solvers (Z3) and sequential verification tools (BOOGIE, with specifications in SPEC#).**

I. 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 [27], and in some cases, like the new semantics of $C{+}{+}$, the semantics is *not even well-defined* when data-races are present [12], [10]. 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.

For sequential programs, one of the most attractive ways of writing reliable code is through the extensive use of *specification languages for writing contracts.* Formal contracts, which were pioneered by the Eiffel framework [28], allow programmers to write pre-conditions, post-conditions, and invariants for methods. The Java Modeling Language (JML) [24] for Java and the SPEC# language [8] for C# from Microsoft espouse the same philosophy of contract-based programming. Both JML and SPEC# are specification languages that have been heavily used in software development and testing. The specifications, apart from serving as formal documentation, are compiled into *runtime* checks in the testing phase, to catch bugs early, as soon as the program strays from the intentions and assumptions stated in the contracts. Contracts can also be subject to *verification* using modern SMT-solvers (satisfiability-modulo theory solvers) to prove them correct. The SAL annotation language and static verifier from Microsoft is an excellent use of this technology, where an annotation language written accompanied by a static checker has been used to eradicate buffer overflows from extremely large code-bases using programmer-written and automatically derived annotations (on the order of ∼400K annotations!) [20].

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.

ACCORD:

In this paper, we ask the following question: What is an appropriate *contract* language for proving race-freedom? We propose an annotation language ACCORD (Annotations for Concurrent Co-ORDination), which is a specification language that captures *thread contracts*, and argue that this language is adequate for documenting code to prove race-freedom. We build both runtime testing and verification techniques to ensure annotation correctness.

The philosophy behind ACCORD specifications is to document *memory-access contracts* between threads. When several threads are spawned, 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 access only when possessing an associated lock. The key idea is that these kind of thread contracts are extremely natural to write (indeed, the programmer clearly knows this when they try to write race-free programs), and moreover, is sufficient to argue and establish that the program is race-free.

In this paper, we develop the ACCORD annotation framework for a restricted class of array-based parallel programs that employ fork-join parallelism, and study the problem of how to statically verify that the annotations indeed prove the program race-free (the annotation adequacy). Moreover, we propose both dynamic and static approaches to verifying that the annotations are indeed correct— i.e. the program satisfies the annotations. The latter annotation correctness phase can be achieved either dynamically by compiling them to *runtime* assertions that can be checked during testing, or by static

program verification techniques using SMT solvers. [1]

***Annotation adequacy and correctness*:**
Hence the verification problems regarding annotations are the following:

- Annotation adequacy: Given a concurrent program P with ACCORD annotations, do the annotations prove that P is free of data-races?
- Annotation correctness: Given a concurrent program P with ACCORD ANNOTATIONS, does the program satisfy the annotations?

Establishing both the above to be true for a program P will, of course, establish that P is race-free.

We show, surprisingly, that annotation adequacy for many array-based fork-join programs can be proved using *constraint solvers*. Intuitively, proving whether the annotations imply race-freedom reduces to the problem of verifying whether two threads are allowed by the annotation 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 with existential quantification) and can hence be handled by constraint solvers based on SMT-solving technology (such as the solver Z3, from Microsoft Research [16]).

Annotation correctness cannot be, of course, completely automated. We show however that ACCORD annotations can be compiled to *assertions* in the concurrent program that can be checked at runtime during testing. Such testing gives increased confidence in the correctness of the annotations themselves, and runtime assertion violations will lead to discovering annotation violations, which in turn can point to races in the program. This mimics similar runtime assertion checking available for sequential contract languages, such as JML and SPEC#.

We also study the problem of statically verifying annotation correctness. Unfortunately, there are no usable concurrent program verification tools based on verification-condition generation that can be put to effective use. However, we show a remarkable result that annotation correctness checking for fork-join programs can be *reduced* to sequential program verification!

More precisely, we show that a concurrent fork-join program with ACCORD annotations can be converted to a sequential program with assertions, and the latter can be verified using existing sequential verification tools (like Boogie, from Microsoft research, that can be used to verify SPEC# specifications). The sequential program is obtained by converting parallel `foreach`-loops to sequential `for`-loops, and the annotations are converted to sequential assertions. The sequentialized program explores only *one interleaving* of the concurrent program (on any input).

The soundness of the above reduction to sequential verification is very subtle, and in fact holds *only if the annotations imply race-freedom* (i.e. provided the annotation adequacy phase passes). We use a circular argument to show that if the sequential program satisfies its annotations, and the annotations imply race-freedom, then the concurrent program cannot expose more behaviors than its sequentialization, and hence the parallel program itself must be race-free. This argument is the first of its kind that we know of, and we think this is a classic *compositional assume-guarantee circular reasoning* [22] argument for proving race-freedom.

In summary, ACCORD annotations provide a language for the programmer to express thread contracts for memory-accesses of individual threads. Whether the annotations imply race-freedom or not can be verified using constraint-solvers and SMT-solvers like Z3 [16]. Annotation correctness can either be checked dynamically using runtime assertions or verified statically using manually-aided sequential verification, utilizing tools such as Boogie [9].

***Evaluation*:** We illustrate our framework using several non-trivial *data-parallel algorithms* that employ fork-join concurrency with extremely intricate sharing of data. These 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) example from the Java Grande benchmark suite [38] that uses an intricate checker-board separation of a 2D-array combined with barriers, a Montecarlo algorithm that uses locks as well as separation to achieve race-free parallelism, as well as several other parallel programs. We also show examples (such as sparse matrix multiplication) that involve concurrent sharing using *indirection*. We also include in our suite buggy variants of the above correct programs, which at first glance seem correct but have subtle errors.

Our main aim is to show that ACCORD specifications are extremely *natural* to write by the programmer, as it captures the sharing strategy directly in a simple logical language. Furthermore, we show that constraint-solvers such as Z3 can easily and automatically prove that the annotations imply race-freedom. The programmer can then gain confidence that the program satisfies the annotations using runtime assertion checking.

In order to show that ACCORD annotations are expressive enough to prove race-freedom, we also show that the static verification of annotation correctness can be achieved for the above programs. We sequentialize the parallel programs, as described above, and use BOOGIE [9] (a sequential verification tool) to prove that all the above (correct) programs conform to their annotations. Some correct programs were in fact *automatically* proved correct, and a few of them needed some manual help in terms of loop-invariants.

While there are limitations to our current framework (for instance, our annotations do not currently handle dynamically allocated heap structures), we emphasize that these are obstacles that can be overcome with richer annotation logics and automated verification tools (for instance, heap structures can be segregated using decidable fragments of separation

[1] We use the terms 'concurrent' and 'parallel' interchangably although a distinction is generally made in literature which is not important in the context of this work.

logic [35] or graph logics in combination with arithmetic and other logical theories [23]).

***Summary*:** Our main thesis is that an annotation language that allows the programmer to specify memory locations accessed by each thread, logically and declaratively, greatly helps in verifying race-freedom, and decomposes the race-freedom verification for concurrent programs into a sequential verification problem and a logical constraint satisfaction problem. The considerable strides of advance that are being made in the latter two areas will yield increasingly practical and largely automated verification tools for race-freedom using our framework. In fact, even if the sequential verification phase is not performed, using existing testing and bug-finding techniques on the sequential program for assertion violations will go a long way in eradicating races.

Therefore we develop an annotation language and accompanying proof mechanisms, and using a class of parallel programs, show that a programmer can give simple and natural annotations, and prove race-freedom using automatic constraint-solvers (SMT solvers) and sequential program verification tools. The salient aspect of our work is that we do not place restrictions on programs, but rather allow programmers to write their code as they wish, provided they can specify their sharing strategy using ACCORD annotations.

II. An Illustrative Example

In this section, we provide an overview of ACCORD approach using a parallel matrix multiplication algorithm. Listing 1 shows the implementation of matrix multiplication 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*:** The annotation (line 8) has a `reads` clause and a `writes` clause, 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. Note that the annotation is allowed to refer to 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

```
28 void mm (int[m,n] A, int[n,p] B) {
29   foreach (int i := 0; i < m; i:=i+1)
30     reads A[i,x],B[x,y],C[i,y] writes C[i,y]
31     where 0 <= x and x < n
32         and 0 <= y and y < p  {
33     for (int j := 0; j < n ; j:=j+1)
34       foreach (int k := 0; k < p; k:=k+1)
35         reads A[i,j],B[j,k],C[i,k]
36         writes C[i,k] {
37         C[i,k] := C[i,k] + (A[i,j] * B[j,k]);
38 } }    }
```

***Specifying arithmetic and logical constraints*:** Observe that the annotation at line 3 also introduces auxiliary variables (x and y); these are *implicitly universally quantified*. This example also demonstrates the usage of a `where` clause (line 4 and 5). A `where` clause constrains (specifies bounds for) the variables in `reads` and `writes` clauses. The annotation at line 3 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 coordination (sharing strategy) among threads, and indeed, the programmer ought to have this separation of memory in mind when reasoning about race-freedom. The programmer providing such annotations greatly simplifies the problem of proving race-freedom. In ACCORD framework, proving that the program is race-free involves two phases: annotation adequacy and annotation correctness.

***Proving the contract implies race-freedom (annotation adequacy)*:** We *automatically* generate a logical formula from the annotation (and not the program) that serves as a verification condition for race-freedom. Specifically, the formula is a conjunction of two sub-formulas: (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 §V for details on generating the formula). The negation of such a formula is provided to Z3. If the formula can be satisfied, there is a race in the program. Otherwise, the program has been proved to be race-free, assuming the annotation is indeed correct.

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.

***Proving the program conforms to the contract (annotation correctness)*:** The second phase of verification requires checking whether the concurrent program satisfies its ACCORD specification. Specifically, 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 can either use concurrent testing tools, which provide some assurance for race-freedom but require no programmer intervention, or sequential verification tools, which prove race freedom but may require the programmer to provide additional invariants. The common

step for both methods is to specialize the annotation for each thread and insert them as assertions in the thread body.

For sequential verification, we additionally *transform* the concurrent program to a sequential program and use BOOGIE to prove properties by inferring invariants and checking Hoare triples (see §VI).

***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], C[i] writes C[i]`. In the annotation adequacy phase, the corresponding formula for showing race-freedom is fed to Z3, which succeeds in proving that the formula is satisfiable. The proof indicates the presence of a data race in the program (there can be two threads writing to C[i]).

III. 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 an annotation language built into the syntax.

$\langle pgm \rangle$ ::= $\langle decl \rangle^* \langle fun \rangle^*$
$\langle fun \rangle$ ::= $\langle type \rangle\ f(\langle decl \rangle^*)\langle \mathbf{annot} \rangle\langle \mathbf{fannot} \rangle\{\langle stmt \rangle\}$
$\langle stmt \rangle$ ::= $\langle decl \rangle \mid \langle loc \rangle := \langle expr \rangle \mid$
`if` $\langle bexpr \rangle$ `then` $\langle stmt \rangle$ `else` $\langle stmt \rangle \mid$
`skip` | `return`$\langle expr \rangle \mid \langle stmt \rangle; \langle stmt \rangle \mid$
`while` $\langle bexpr \rangle$ *do* $\{\langle stmt \rangle\} \mid \langle parstmt \rangle \mid$
`for`$(\langle loc \rangle := \langle expr \rangle; \langle bexpr \rangle; \langle stmt \rangle)\{\langle stmt \rangle\} \mid$
`acq` $l \mid$ `rel` l
$\langle parstmt \rangle$ ::= `thread`$\langle \mathbf{annot} \rangle\{\langle stmt \rangle\} \mid$
`foreach`$(\langle loc \rangle := \langle expr \rangle; \langle bexpr \rangle; \langle stmt \rangle)[\langle \mathbf{annot} \rangle]$
$\{\langle stmt \rangle\} \mid \langle parstmt \rangle$ `with` $\langle parstmt \rangle$
$\langle loc \rangle$ ::= $i \mid i[\langle aexpr \rangle^*]$
$\langle expr \rangle$ ::= $\langle aexpr \rangle \mid \langle bexpr \rangle \mid f(\langle expr \rangle^*)$
$\langle bexpr \rangle$ ::= `true` | `false` | $\langle aexpr \rangle \langle rop \rangle \langle aexpr \rangle \mid$
$\langle bexpr \rangle$ `or` $\langle bexpr \rangle \mid \langle bexpr \rangle$ `and` $\langle bexpr \rangle \mid$ `not` $\langle bexpr \rangle \mid$
$\langle aexpr \rangle$`implies`$\langle aexpr \rangle$
$\langle aexpr \rangle$ ::= $c \in \mathbb{N} \mid \langle loc \rangle \mid \langle aexpr \rangle \langle aop \rangle \langle aexpr \rangle$
$\langle decl \rangle$ ::= $\langle type \rangle i$
$\langle type \rangle$ ::= `int` | `bool` | `char` | `void` | `lock` | `lock`$[\langle aexpr \rangle^*]$ |
`int`$[\langle aexpr \rangle^*]$ | `bool`$[\langle aexpr \rangle^*]$ | `char`$[\langle aexpr \rangle^*]$

$\langle \mathbf{annot} \rangle$ ::= `reads` $\langle cloc \rangle$`writes` $\langle cloc \rangle$[`assumes` $\langle bexpr \rangle$]
$\langle cloc \rangle$ ::= $\langle cloc \rangle$ `where` $\langle bexpr \rangle \mid \langle cloc \rangle, \langle cloc \rangle \mid$
$\langle loc \rangle \mid \langle cloc \rangle$ `under lock` l
$\langle \mathbf{fannot} \rangle$ ::= [`requires` $\langle bexpr \rangle$][`ensures` $\langle bexpr \rangle$]

i, f, l are identifiers
$\langle aop \rangle$: arithmetic operators, like $+$, $-$, $*$, $/$, $\%$ (modulo), etc.
$\langle rop \rangle$: 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, acquiring and releasing a lock, and sequential loops, or a parallel statement. A parallel statement can be a `foreach` loop, which forks a separate thread for executing each different iteration of the loop. All the threads spawned at this point must finish before the subsequent statement is executed. Hence `foreach` loops implicitly give fork-join parallelism. The `with` construct is a parallel composition operator, and also implicitly defines fork-join parallelism.

The `foreach` construct can be optionally augmented with an *annotation* ($\langle annot \rangle$). Such a parallel-loop annotation declares, for each thread spawned at this point, the set of memory locations it will access. The `under lock` clause expresses the pattern that a memory location is accessed only when a lock is held by the executing thread. The annotation language has an `assumes` clause that allows making general assumptions on the state of the program variables when the forking happens.

Function declarations can also be annotated; these annotations include read/write annotations (the memory locations read and written by the function) as well as pre-conditions (`requires` clause) and post-conditions (`ensures` clause) (see the $\langle fannot \rangle$ definition). Post-conditions are also allowed to use a special variable called `\result` which refers to the value returned by the function.

The annotation language has no *execution semantics*, and is designed to (a) help the programmer annotate the precise parts of the memory accessed by each thread, along with the assumptions it makes, and (b) be sufficient to prove that the program is race-free.

Let us assume that there is a `main` method, where the program starts, and that there are no calls to this method. 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 a static region-based alias analysis integrated into our framework. 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: loops, conditionals, etc. have the normal semantics, calls to functions are call-by-value, and all assignment statements are meant to occur *atomically* even if they involve multiple reads and writes. The semantics for the `foreach` construct is that it forks many threads, one for each loop value of the loop-index in the range, which execute the body of the loop. 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 acquiring and releasing locks is also the usual one (we do not handle re-entrant locks in this exposition).

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 sequential semantics is the standard one (see [4] for instance). Note that races on locks are not considered data-races. Programming languages such as Java and C++ have, of course, much more complex semantics. Furthermore, the actual memory model assured by such languages is much weaker than the sequentially-consistent model, but in the absence of data-races, the memory model assures that the programs will be executed in a sequentially-consistent manner. Our goal is to write programs with annotations that we can use to help prove them data-race free.

IV. 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) [32], a parallel implementation of the MonteCarlo simulation program (only the parallel component) from the Java Grande benchmark suite [38], and a parallel implementation of LU Factorization algorithm. Note that the programs have been annotated by the authors.

A. 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 black and red (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, followed by all the black elements.

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 (double[m,n] A, double w) {
2
3   //update red
4   foreach (int id := 1; id < m; id:=id+1)
5    reads 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 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     assumes ((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  foreach (int id := 1; id < m; id:=id+1)
21   reads A[id,j], A[id-1,j], A[id+1,j], A[id,j-1],
22     A[id,j+1] writes A[id,j]
23   where 0 <= j and j < n and ((id+j) % 2 = 1) {
24
25    foreach(int k := 1+(id%2); k < n; k := k+2){
26      reads A[id,k], A[id-1,k], A[id+1,k],
27        A[id,k-1], A[id,k+1]
28     writes A[id,k]
29     assumes ((id + k) % 2 = 1) {
30
31      A[id,k] := (1 - w) * A[id,k] + w * 0.25 *
32       (A[id-1,k]+A[id+1,k]+A[id,k-1]+A[id,k+1]);
33 } } }
```

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 implicitly universally quantified, 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 but not shown due to limited space) reads both red and black elements, and updates black elements only. Therefore, the annotations include the arithmetic constraint $((x + y)\%2 = 1)$ which defines the black elements.

B. Quicksort

We implement a race-free, fully parallel algorithm for quicksort [32]. 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. This is done by using a parallel rank (prefix sum) algorithm.

Listing 3 and 4 shows our implementation. The main method takes an array $A[n]$ as input and sorts its contents. Due to limited space, we have not shown the method `rel_pos_rec`, but it shows a similar parallelism as the `write_pos_rec`.

Listing 3 Quicksort algorithm with parallel partitioning.

```
1  void qsort (int[n] A, int i, int j) writes A[k]
2      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    thread writes A[k] where (i<=k and k<p_index)
9           assumes p_index >= i;
10       { qsort(A, i, p_index) ; }
11   with
12   thread writes A[k] where (p_index < k and k < j)
13          assumes p_index < j;
14       { qsort(A, p_index + 1, j) ; };
15 }
16
17 int dyn_partition(int[n] A,int pivot,int i,int j)
18   ensures i <= \result and \result < j {
19
20   int[n] temp, B;
21   int smalls := rel_pos_rec(A,temp,pivot,i,j);
22   write_pos_rec(A,B,temp,pivot,i,j,0,smalls);
23   A := B;
24   return smalls;
25 }
```

In addition to being a long, complex program, it highlights some interesting features of our annotation language. The `thread` statement can be annotated (lines 8 and 12 in Listing 3), just like the `foreach` statement, and the annotation specify the set of memory locations read and written by the thread. This example also requires annotating a method (lines 1-2 and lines 17-18 in Listing 3), specifying a post-condition for the `dyn_partition` method using an `ensures` clause that says that the chosen pivot lies within the range defined by i and j. Method annotations, which include `reads` and `writes` clauses, are important for the annotation correctness phase (validation of annotation against the program). The quicksort program also shows that recursive partitioning can be handled by our annotation language in addition to iterative partitioning of data.

C. 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.

Listing 4 Quicksort algorithm with parallel partitioning (Contd..)

```
26 void write_pos_rec(int[n] A,int[n] B,int[n] temp,
27       int pivot,int i,int j,int pos1,int pos2){
28   int divide := (i+j)/2;
29   if (j-i < 2) {
30     if (A[i] <= pivot) B[pos1] := A[i];
31     else B[pos2] := A[i];
32     return;
33   };
34
35   thread reads A[k], temp[k]
36      where (i <= k and k < divide) writes B[k]
37      where (pos1 <= k and k < pos1+temp[divide])
38      or (pos2<=k and k<pos2+divide-i-temp[divide])
39      {
40       write_pos_rec(A, B, temp, pivot, i, divide,
41         pos1, pos2); }
42   with
43   thread reads A[k], temp[k]
44     where (divide <= k and k < j) writes B[k]
45      where (k >= pos1+temp[divide] and h < pos2)
46      or (k >= pos2 + divide - i - temp[divide])
47      {
48       write_pos_rec(A,B,temp,pivot,divide,j,
49         pos1 + temp[divide],
50         pos2 + divide - i - temp[divide]);
51 } }
52
53 int rel_pos_rec(int[n] A, int[n] temp,
54           int pivot, int i, int j) {
55   if (j-i < 2) {
56     if (A[i] <= pivot) return 1;
57     else return 0;
58   }
59
60   int divide := (i+j)/2;
61   int a1, a2;
62
63   thread reads A[k] where (i <= k and k < divide)
64     writes temp[k] where (i <= k and k < divide)
65        { a1 := rel_pos_rec(A, temp, pivot, i,
66          divide); }
67   with
68   thread reads A[k] where (divide <= k and k < j)
69     writes temp[k] where (divide <= k and k < j)
70        { a2 := rel_pos_rec(A, temp, pivot,
71          divide, j); };
72
73   temp[divide] := a1;
74   return a1 + a2;
75 }
```

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 access is protected by a lock. After the parallel phase, the results are reduced in a sequential fashion.

Listing 5 shows a simplified version of the program for illustration. Some methods have not been shown due to limited space. The main method 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 20 before writing to

Listing 5 Simplified version of the MonteCarlo simulation code from Java Grande.

```
1  void main(int[nTasks] tasks) {
2    foreach (int i:=0;i<nthreads;i:=i+1)
3      reads next under lock gl,tasks[k]
4      writes next,results[j] under lock gl
5      where (i*slice)<=k and k<((i+1)*slice) and
6      slice := (nTasks + nthreads-1)/nthreads {
7
8      appRun(i, tasks);
9  } }
10
11 void appRun(int id, int[nTasks] tasks) {
12   int slice := (nTasks+nthreads-1)/nthreads;
13   int ilow := id*slice;
14   int iupper := (id+1)*slice;
15   if (id = nthreads-1) iupper := nTasks;
16
17   for(int run:=ilow; run<iupper; run:=run+1){
18     int result := simulate(tasks[run]);
19
20     acq gl;
21     next := next + 1;
22     results[next] := result;
23     rel gl;
24 } }
```

the shared variables $next$ and $results$, and then releases the lock at line 23.

D. LU Factorization

LU Factorization is an important kernel in numerical analysis and scientific computing applications. It is often used to solve systems of linear equations. LU Factorization can be computed using Gaussian elimination. The method iteratively eliminates one unknown until the solution for a single unknown is found. This solution is then substituted backwards to solve the system of equations. In general, row interchanging (pivoting) may be required to ensure that LU Factorization exists.

The code in Listing 6 shows a parallel implementation of LU Factorization method. The code assumes the pivot has been already computed. The program takes a matrix $A[m,n]$ as input, and updates the elements in the matrix A. The outermost loop (line 4) iterates over each column (unknown variable) of the matrix, eliminating one unknown in each iteration. The `foreach` loop (line 5) iterates over the columns to the right in steps of size two, assigning two columns to a single thread. Note that this assignment of two columns to a thread is a variant of the standard algorithm; we use this to highlight a feature of our language, as explained below.

Two important things to note in this example are:

1) The region of computation is shrinking for each later iteration. Hence, the columns (set of memory locations) assigned to a thread are dynamic. This differentiates our approach from region-based approaches [11], which tend to be static in terms of sharing strategy. The first conjunct in the `where` clause captures the dynamically shrinking region.

Listing 6 LU Factorization with 1-D Column Agglomeration

```
1  void par_lu (int A[i,j])
2  {
3    //iterate over columns
4    for (int c = 0; c < j; c++)
5      foreach (int k =0; k < (j - c)/2; k++)
6        reads A[m,n], A[m,c], A[c,n] writes A[m,n]
7        where c <= m and m < i and
8          c + 1 + k*2 <= n and n < c + 1 + (k+1)*2 {
9
10       lumethod(A, c+1+ k*2, c+1+ (k+1)*2, c);
11 }   }
12
13 void lumethod (int A[i,j], int a, int b, int c)
14 {
15   if (b > j) b = j;
16   for (int x = a; x < b; x++)
17     for (int y = c+1; y < i; y++)
18       A[y,x]=A[y,x]-(A[y,c]/A[c,c]) * A[c,x];
19 }
```

2) While previous examples split the matrix either row-wise or column-wise, in this example the matrix is partitioned such that each thread is assigned two columns. In order to capture the memory locations written by each thread, an arithmetic constraint (lines 7-8) is specified over the column index n. Note that the constraint is a function of the loop variable k.

Each iteration of the `foreach` loop calls a function which performs the updates for all elements within the columns assigned to a thread. In order to avoid making the example more verbose than required, we do not exploit the parallelism available within this method.

V. 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 annotations which is then checked by Z3, an SMT solver from Microsoft Research. Intuitively, the verification condition is constructed such that it is satisfiable if the regions read and written by two different threads have some common memory location. Hence, if the condition is satisfied, then a satisfying solution gives **valuable debugging information**: the two threads and a memory location that they are allowed to access according to the contract, where one of the access is a write. On the other hand, if the condition is unsatisfiable, we are assured that the regions are disjoint according to its declared contract.

Consider a parallel loop of the form: `foreach`(i:=1, cond(i), i:=i+1) with an annotation that associates a read-set of locations $R(i)$ and a write-set of locations $W(i)$, for each loop index i. The verification condition is a logical formula that expresses the following: does there exist two distinct, valid indices i_1 and i_2 of the loop, such that (a) the read-set of thread i_1 and the write-set of thread i_2 intersect, (i.e.

$(R(i_1) \cap W(i_2) \neq \emptyset)$, or (b) the write-sets of threads i_1 and i_2 intersect, (i.e. $(W(i_1) \cap W(i_2) \neq \emptyset)$.

We assume that the annotation of a parallel loop expresses that the loop-index is within the array bounds, if it is required for proving race-freedom.

***Translating a read/write annotation to logic*:** Let `reads` φ (or `writes` φ) be a read (or write) annotation. Consider an array $A[\vec{x}]$ and a loop-index variable i. We can now write a logical formula $Conforms(A, \vec{x}, i, \varphi) = [\![A[\vec{x}] \in \varphi_i]\!]$ that expresses whether the indices $\vec{x}$ satisfy the annotation φ for loop-index i. Intuitively, this translation from the annotation results in a constraint on the indices using arithmetic and logical constraints, parameterized by the loop-index.

$$\begin{array}{rcl}
[\![A[\vec{x}] \in b_1, b_2]\!] & \mapsto & [\![A[\vec{x}] \in b_1]\!] \vee [\![A[\vec{x}] \in b_2]\!] \\
[\![A[\vec{x}] \in b \texttt{ where } \phi]\!] & \mapsto & [\![A[\vec{x}] \in b]\!] \wedge \phi \\
[\![A[\vec{x}] \in A[\vec{e}]]\!] & \mapsto & x_1 = e_1 \wedge \cdots \wedge x_n = e_n \\
[\![A[\vec{x}] \in B[\vec{e}]]\!] & \mapsto & \mathit{false}\ (\mathit{where}\ A, B\ \mathit{do\ not\ alias}) \\
[\![A[\vec{x}] \in b \texttt{ under lock } l]\!] & \mapsto & \mathit{false}
\end{array}$$

For instance, consider the first `writes` annotation on line 3 in Listing 1. Then, for a parallel iteration with loop-index i_1,

$[\![C[i,k] \in (C[i_1, y] \texttt{ where } 0 \leq x \wedge x < n \wedge 0 \leq y \wedge y < p)]\!]$
$= [\![C[i,k] \in C[i_1, y]]\!] \wedge 0 \leq x \wedge x < n \wedge 0 \leq y \wedge y < p$
$= (i = i_1 \wedge k = y \wedge 0 \leq x \wedge x < n \wedge 0 \leq y \wedge y < p)$.

Note that any access that is protected by a lock is data-race-free by definition. It is possible that two different threads access the same location under two different locks but this can identified using a syntactic check on the annotation. The above definition of $[\![\ ]\!]$ will also be used in the next phase (see §VI).

***The non-interference constraint*:** We are now ready to define the constraint that is satisfiable iff there is a data-race implied by the annotations. Consider a parallel loop with a read annotation r and a write annotation w and an assumption ψ. Then we generate the following constraint:

$$\exists i_1, i_2.\ [\ (i_1 \neq i_2) \wedge \psi \wedge \bigvee_{A \in fa(r) \cap fa(w)} (Conforms(A, \vec{x}, i_1, r) \wedge Conforms(A, \vec{x}, i_2, w))]$$

In the above, $fa(r)$ and $fa(w)$ are the arrays/variables specified in each annotation r and w, respectively. The formula hence says that are two distinct parallel indices and some variable/array that is read by thread i_1 and written to by thread i_2.

We write a similar constraint for checking write-write races:

$$\exists i_1, i_2.\ [\ (i_1 \neq i_2) \wedge \psi \wedge \bigvee_{A \in fa(w)} (Conforms(A, \vec{x}, i_1, w) \wedge Conforms(A, \vec{x}, i_2, w))]$$

The above formulas, for each parallel-loop annotation and each method annotation, are translated into the syntax accepted by Z3, and fed to it. The constraints are all unsatisfiable iff the annotations imply race-freedom. If a constraint is satisfied, then Z3 usually returns a satisfying assignment which is a witness to the fact that the annotation does not ensure race-freedom.

VI. Checking Annotation Correctness

The objective of this phase is to verify that a parallel program conforms to its thread contracts.

***Using Concurrent Testing Tools*:** One approach is to use existing concurrent testing tools such as Chess, Java PathFinder. This is done by specializing the annotation to each thread and translating them to runtime assertions which are then inserted in the thread body. The program with annotations is then tested with one of these tools. This provides assurance that the program meets the annotations for some inputs and for some schedules. Note that this does not require programmer intervention.

We now transform the concurrent ACCORD annotations to assertions on the concurrent program. Given a read or write access to an array A at indices $\vec{i}$ by the program (which may be empty — plain variables are treated as zero-dimensional arrays), we wish to ensure the array access is within the declared read or write region from the surrounding annotation. We annotate the array access with an assertion that demands that the indices are within the bounds, using a first order statement.

First, we remove the ACCORD annotations from P. For any annotation a that encloses (i.e. occurs in the scope of) the statement, we first cache right before the annotation (i.e., store in an auxiliary variable) the current values of the program variables the `where` clause refers to. Let the cached program variables be denoted as PV and the free variables be $AuxV$.

Consider a read (or write) access statement to an array $A[\vec{x}]$. For every read (or write, respectively) annotation that encloses the statement, we insert an assertion corresponding to this annotation. Let us first assume that this annotation has the program variables mentioned in the `where` clause, *except the indices of the parallel loops the statement is within*, replaced using the cached copies (the cached copies won't change during the portion of the program from the annotation to the statement). Let `reads` φ (or `writes` φ) be such an annotation; then we define the corresponding assertion "`assert` $[\![A[\vec{x}] \in \varphi]\!]$". Intuitively, this assertion translates the constraint on the indices to arithmetic and logical constraints, parameterized by the loop-index.

***Using Sequential Verification Tools*:** Another technique is to phrase this as a sequential verification problem and use existing tools such as BOOGIE. Intuitively, checking whether each parallel iteration of a `foreach` loop conforms to reading and writing to only the locations that the annotation specifies, does not really require us to execute the various iterations in parallel, and surprisingly can be achieved by considering a sequential execution of the parallel program. However, we aim for a generic construction that creates a single sequential program for all threads at all forking points at once, along with procedural pre- and post-conditions. The soundness of such a sequentialization is subtle, because if the threads interact, a sequentialization may satisfy an annotation while the parallel program may not.

For instance, consider the following synthetic program:

```
int A[4];
A[0]=1; A[1]:=0; A[2]:=0;
foreach(i:=1; i<3; i:=i+1)
  reads A[k] writes A[i] {
  A[A[i-1]]:=A[i-1]+1;
}
```

The annotation says that the i'th parallel iteration may write only to A[i], which is blatantly false as, for instance, the thread corresponding to $i = 2$ may first execute and, since A[1]=0, it sets A[0]:=1. However, if we sequentialize the program by replacing the foreach-loop above by a for-loop, the annotation is indeed correct, as thread with $i = 1$ executes first, executes `A[1]:=2`, and followed by the thread with $i = 2$, which writes to $A[2]$ only.

Our crucial observation is that the only reason that this analysis of the sequentialization fails is because the *annotation does not imply race-freedom* (the annotation says that thread with $i = 1$ may write to A[1] and thread with $i = 2$ may read A[1]). If we assume that the annotation adequacy phase has passed and thus implied race-freedom, then analyzing a sequentialized program against the annotation suffices. Our soundness argument hence depends on the success of the adequacy phase. Intuitively, if the annotation implies race-freedom, then each parallel thread, as long as it conforms to the annotation, cannot affect the other thread, and hence cannot *help* the other thread satisfy the annotation.

The translation to a sequential program is fairly involved and we describe it in the following steps. Consider a concurrent program P with ACCORD annotations; we describe its translation to a sequential program P_{seq} in three steps.

***Step 1*:** We replace all `foreach` statements with `for` statements. Furthermore, every statement of the kind *parstmt1* `with` *parstmt2* is transformed to *parstmt1*; *parstmt* (i.e. the parallel composition operator is replaced by a sequential composition operator). The resulting program is completely sequential, and obviously, its behaviors are a subset of the behaviors of the concurrent program.

In addition, we replace declaration of a lock by a declaration of an integer variable with the same identifier, and initialized with 0. Also, we replace every statement of the form (`acq` l) and (`rel` l) by a statement that increments and decrements, respectively, the corresponding integer variable l by one.

***Step 2*:** We retain the pre- and post-conditions of methods in the sequential program. These assertions will be verified in the sequential verification phase, and will interact and aid in proving the assertions obtained by translating the concurrent ACCORD annotations.

***Step 3*:** We also insert sequential annotation (assertion) to ensure that access are within the bounds declared in the contracts. These assertions are the same as $[\![]\!]$ except that in order to handle locks, we introduce $\langle\!\langle\rangle\!\rangle$ that modifies $[\![]\!]$ (see §V) in the following way:

$$\begin{array}{rcl} \langle\!\langle A[\vec{x}] \in b \texttt{ under lock } l \rangle\!\rangle & \mapsto & \langle\!\langle A[\vec{x}] \in b \rangle\!\rangle \wedge l > 0 \\ \langle\!\langle A[\vec{x}] \in c \rangle\!\rangle & \mapsto & [\![A[\vec{x}] \in c]\!] \\ & & (\texttt{c } \textit{is not } \texttt{b under lock}) \end{array}$$

Apart from the above assertions, any `assume` φ annotations are also added as assertions of the form `assert` φ. We insert the following assertion at the beginning of each thread and `foreach` loop body for every lock l in the enclosing annotation: `assert` $l = 0$, and the following assertion before each lock l release: `assert` $l > 0$. Also, we can not assume before a lock l acquire that the location protected under l has a particular value. Therefore, just after a lock acquire we assign the location protected under the lock (according to the annotation), a nondeterministic value which can be represented as a function that returns unknown value in BOOGIE as well as Z3.

***Example*:** Assume a program with an assignment to the memory location $A[i, r+1, s-1]$ and assume that there is a write annotation "$A[i, j, k]$ `where` $j\%2 = p$ `under lock` pA" enclosing the statement, where p, k are references to program variables, i is the loop-index, and j is a free variable. We cache the values of p, k by inserting statements of the form $p' := p$ just before where the annotation is specified. Then, by applying the rules described above, we obtain a formula:

$$\begin{array}{l} \langle\!\langle A[i, r+1, s-1] \in A[i_1, j, k'] \texttt{ where } j\%2 = p' \texttt{ under lock } pA \rangle\!\rangle \\ = \langle\!\langle A[i, r{+}1, s{-}1] \in A[i_1, j, k'] \texttt{ where } j\%2{=}p' \rangle\!\rangle \wedge pA > 0 \\ = [\![A[i, r{+}1, s{-}1] \in A[i_1, j, k']]\!] \wedge\ j\%2{=}p' \wedge pA > 0 \\ = (i{=}i_1 \wedge r{+}1{=}j \wedge s{-}1{=}k' \wedge j\%2{=}p' \wedge pA > 0). \end{array}$$

We then add this as an assertion with the free variable j existentially quantified. Eliminating the existential quantifications using equality constraint on j gives the following statement:

`assert`$(i{=}i_1 \wedge (s{-}1){=}k' \wedge (r{+}1)\%2{=}b' \wedge pA > 0\ )$;

which is inserted just before the assignment in the sequential program.

Consider the matrix multiplication program given in Listing 1. The sequential program that corresponds to it is given in Listing 7. The first three assertions (lines 6-8) correspond to the first annotation and the accesses to A, B and C, while the last three assertions (lines 10-12) correspond to the second annotation.

Listing 7 Sequential program corresponding to the parallel implementation of matrix multiplication in Listing 1

```
1  void mm (int[m,n] A, int[n,p] B) {
2    for (int i := 0; i < m; i:=i+1) {
3      n':=n; p':=p;
4      for (int j := 0; j < n ; j:=j+1)
5        for (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 } }    }
```

In Listing 8 we present the sequentialization of the MonteCarlo simulation (§IV) since it illustrates how locks are handled in the annotation correctness phase. Note how the location protected by a lock (*next*) is assigned a nondeterministic value on Line 15. This models the nondeterminism due to interleaving of threads accessing a shared location.

Listing 8 Sequentialization of the MonteCarlo simulation code.

```
1  void appRun(int id, int[nTasks] tasks) {
2    assert gl = 0;
3    int slice := (nTasks + nthreads-1)/nthreads;
4    int ilow := id*slice;
5    int iupper := (id+1)*slice;
6    if (id = nthreads-1) iupper := nTasks;
7
8    for(int run:=ilow; run<iupper; run:=run+1){
9      assert i=id && run=k && (i*slice) <= k
10      && k < ((i+1)*slice)
11      && slice := (nTasks+nthreads-1)/nthreads;
12     int result := simulate(tasks[run]);
13
14     gl := gl+1; //acq gl;
15     //assign a nondeterministic value
16     next := input();
17
18     assert gl>0;
19     next := next + 1;
20     results[next] := result;
21
22     assert gl>0;
23     gl := gl-1; //rel gl;
24 } }
```

We can now argue the correctness of the sequentialization:

Theorem 1: Let P be a concurrent program with ACCORD annotations, and P_{seq} be its sequentialization with the appropriate assertions as described above. Further, assume that the annotations of P imply race-freedom. The concurrent program P satisfies its annotations iff the the sequential program P_{seq} satisfies its assertions.

***Proof Sketch*:** First, let us assume that there are no locks in the program; we will argue this case first as it captures the crux of the argument. Note that we are *not* assuming that P is race-free, but simply that its annotations imply race-freedom.

One direction is easy: if the concurrent program P satisfies its ACCORD annotations, then it is easy to see that P_{seq} generates only a subclass of behaviors of the concurrent program, and since we rewrite the annotations as local assertions (using caching of variables and instantiating the constraint on the particular loop), they will hold in the sequential program as well.

The other direction, soundness, is harder, and crucially utilizes the assumption that the ACCORD specifications of P imply race-freedom. By way of contradition, let P be a concurrent program without locks that does not satisfy its annotations and assume its sequentialization P_{seq} does satisfy its assertions.

The high-level argument is an involved and subtle argument, and can be summarized as follows. consider the smallest prefix $\sigma.e$ of an interleaved execution of P that violates an annotation. Since all events in σ satisfied the annotations, and the annotations imply race-freedom, there could have been no non-trivial interaction between any concurrent threads. Hence the state of the sequential program when it executes e will be the same as that of P when executing e. Since the former satisfies the annotation, so must the latter, which contradicts our assumption.

The above argument just gives the intuition for correctness: a more precise argument must consider the nesting of parallel loops and the precise set of statements concurent with e. A more detailed proof follows here.

A. Gist of Proof of Theorem 1

Let us assume that P has only `foreach` loops, and no `with` construct (the proof including the `with` construct is similar).

Let $\rho.e$ be a *shortest* (in length) interleaved run of P such that the last event e in $\rho.e$ violates an annotation assumption in P. Assume that this last event e in $\rho.e$ occurs within r `foreach` constructs, say at program points $pc_1, \ldots pc_r$. Now, consider the parallel threads spawned at pc_1. The final event e belongs only to one of these threads, say for loop index i_1, and hence, in $\rho.e$, all other events that occurred in the other threads spawned at pc_1 (call this set F) meet their annotation specification (since they occur in ρ before e and e is the first violation in $\rho.e$). Since the annotation specification implies race-freedom, the events in F cannot causally affect any event in the thread for loop index i_1. By the assumption that $\rho.e$ is shortest, it follows that $\rho.e$ cannot contain any of these events in F. In other words, $F = \emptyset$. Hence $\rho.e$ contains events of only *one* thread spawned at each of the `foreach` loops at $r_1, \ldots, r_k$.

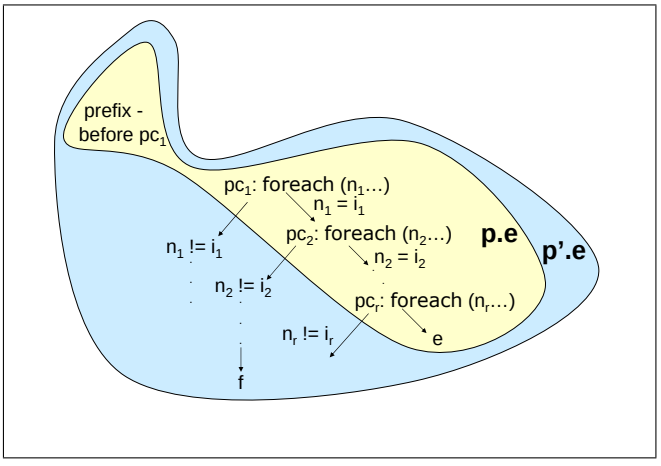

Fig. 1. Proof Sketch.

Now consider the sequentialized run $\rho'.e$ that executes the events in $\rho.e$, precisely ending at event e. Note that both $\rho.e$ and $\rho'.e$ must have done all *completed* `foreach` loops that occurred before pc_1, to completion, but perhaps in different order. However, since the annotations are not violated by the concurrent program in that prefix of the run before pc_1, and

the annotations imply race-freedom, it follows that the run up till pc_1 is actually *deterministic* and hence the run in the sequential program and the concurrent program run will be in the same precise state when they execute the outer-most `foreach` loop at pc_1.

$\rho'.e$ may have events f that are scheduled in threads that are spawned at `foreach` loops at $pc_1, \ldots pc_k$ that are *concurrent* with the thread e belongs to but are scheduled before in the sequentialization (see Figure). However, since the sequential program satisfies the assertions obtained from the annotations, and the annotations imply race-freedom, it follows that these events in $\rho'.e$ cannot affect the state e operates at in any way. Hence after ρ', e necessarily violates the annotation, and hence the corresponding sequential assertion is violated in P_{seq}, which contradicts our assumption that P_{seq} satisfied all its assertions. This contradiction proves the soundness of our reduction— if the concurrent program did not satisfy its annotations, then its sequentialization cannot satisfy its sequential assertions.

Let us now consider locks. Recall that locks are sequentialized introducing a corresponding integer to keep track of the number of acquisitions, and whenever a lock is acquired, all variables protected under the lock in the relevant annotation are set to a nondeterministic value. The argument for correctness of locations that are not protected by locks remains the same. Observe that for locations protected by locks, the sequentialization over-approximates the effect of the interaction by setting these values to arbitrary values. If a program thus sequentialized passes all assertions, it is clear that the concurrent program also does. Note that this proof technique will work only if the locations accessed by a thread does not dynamically change with interactions from other threads. Most programs indeed respect this constraint.

VII. Evaluation

In this section, we describe our experience in augmenting concurrent programs with ACCORD in order to prove race-freedom. We provide statistics of annotation burden and time taken for the two verification phases and show that the ACCORD annotations are short and do not put an undue burden on the programmer, that the annotation schemes can handle complex and realistic programs.

Apart from the examples that we have described in sections §II and §IV (matrix multiplication, SOR, QuickSort and MonteCarlo), we also annotated and checked a number of programs from the Java Grande Forum (JGF) benchmark suite as well as parallel LU Factorization (LuFact). Specifically, we include `series` (which implements Fourier coefficients of a function), `moldyn` (an N-body code modeling interacting particles) and `sparsematmult`. For the `sparsematmult`, we only considered the parallel component. The complex condition describing the indirection in accessing the matrix is provided as a pre-condition of the parallel phase. These programs are parametrized by the number of threads: during execution, the threads divide the data among themselves by computing a non-linear formula over the total number of threads and size of the data. This formula forms the region of computation for each thread.

We were able to annotate these programs and prove them race-free using ACCORD. Table I presents the results of our evaluation. The name and the code size of our benchmarks are given in the first two columns. Note that these are considerably larger programs (up to 1300 LOC).

***Annotations*:** The next four columns (labeled *Annotations*) of Table I provide information about the annotations required for these programs. The third column records the number of total `reads` and total `writes` clauses required to annotate each program. The next column records the number of total auxiliary variables used in these annotations. While auxiliary variable provides the programmer the ability to declare sharing strategy independent of the implementation, On the other hand, it puts an extra burden on the programmer as the constraints on the auxiliary variables need to be specified. The number of `where` clauses in each program are listed in the next column. The last column under *Annotations* lists the number of pre- and post-conditions in the program.

The annotation statistics show that the additional burden on the programmer incurred in writing ACCORD annotations is not much. These results and our experience suggest that the annotations themselves are a natural way to express the parallelization strategy. We believe, ACCORD annotations can simplify both manual and automatic reasoning significantly.

***Checking Annotation Correctness*:** We use the BOOGIE tool to check whether the program conforms to the annotation. Although BOOGIE handles many verification tasks automatically, this phase needs some programmer support to verify certain logical properties about the variables. Specifically, almost all looping constructs required adding loop invariants manually. The number of such manual assertions is recorded in column 6. The next two columns report the time taken to complete the phase of annotation correctness checking (which is minimal), and whether the annotations are proved to be correct by BOOGIE. Note that we do not perform the annotation correctness phase for programs that fail the annotation adequacy phase (see §VI).

***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 first column in the adequacy phase gives the logic used to prove whether the verification condition is satisfiable or not. We use the SMT_LIB notation [34]. Note that most programs are proved race-free using linear integer constraints. Proving the SOR program requires non-linear integer arithmetic logic. The verification condition for `series` benchmark 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 `series`.

The next column reports the time taken by Z3, and whether the annotations implied race-freedom (again, the time taken is

TABLE I
SUMMARY OF RESULTS FROM EVALUATING ACCORD.

	Lines of code	*Annotations*				*Correctness phase*			*Adequacy phase*			Proven Race Free?
		# `reads/ writes` clauses	# Total aux. vars	# `where` clauses	# Pre/ Post cond.	# invar. added	Time taken	Success? (Yes/No)	Logic used	Time taken	Success? (Yes/No)	
MatMult	25	2/2	2	1	-	3	<1s	Yes	QF_LIA	<1s	Yes	Yes
MatMult (buggy)	30	3/3	2	1	-	-	-	-	QF_LIA	<1s	No	No
SOR	45	4/4	2	4	-	6	<1s	Yes	QF_NIA	<1s	Yes	Yes
Quicksort	100	4/7	11	9	0/1	2	2s	Yes	QF_LIA	<1s	Yes	Yes
MonteCarlo	255	1/1	2	1	0/0	8	90s	Yes	QF_UFLIA+MA	<1s	Yes	Yes
LuFact	35	1/1	2	1	0/0	2	<1s	Yes	QF_LIA	<1s	Yes	Yes
LuFact (buggy)	35	1/1	2	1	0/0	-	-	-	QF_LIA	<1s	No	No
`sparsematmult-jgf`	50	1	3	1	6/0	1	2s	Yes	QF_UFLIA+MA	<1s	Yes	Yes
`series-jgf`	800	1/1	3	1	0/0	2	<1s	Yes	QF_UFLIA+MA	<1s	Yes	Yes
`moldyn-jgf`	1300	5/6	4	6	0/0	0	5s	Yes	QF_LIA	<1s	Yes	Yes

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

minimal). 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 §II), 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. This strengthens our belief that using ACCORD specifications to prove race-freedom can help discover bugs due to complex sharing strategies.

VIII. RELATED WORK

There is a rich literature on using type analysis in order to ensure race-freedom [6], [11], [30], [13], [19], [7], [18]. 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 [11] 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 implement, 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 [6] 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 [25] 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.

Recent work has proposed an annotation language for determinism property [14], and employs dynamic exploration of different behaviors of the concurrent program in order to verify it. There is some recent unpublished work [26] close to our approach that proposes *inferring* the read and write regions from loop-free (SPMD) CUDA [3] 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 [35] 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, and is really a useful dialect of first-order (and monadic second-order) constraints on the heap; further, decision procedures for separation logic are based on particular proof systems developed for them, and has not been incorporated within mainstream SMT solvers in order to combine it with other theories such as arithmetic using

standard Nelson-Oppen combinations. Our annotation logic has been developed using explicit first-order constraints, with the separation implicit using the parallel loop-indices, but endowed with arithmetic constraints that allow fine separation of a data-structure. When moving to programs with dynamic data, we certainly envisage using logics for separation.

Havoc [23] is a tool that utilizes a very specific logic for heaps, based on reachability predicates, where constraints can be transformed to queries of SMT solvers and allows combinations with other theories such as arithmetic and uninterpreted functions. This again may be a suitable basis for annotation languages for extending our mechanisms to handle heaps.

There is also 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 [36], vector-clock based algorithms [37], hybrid algorithms [33], [40], Goldilocks [17], and static-analysis algorithms [15].

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 [39]. The primary motivation in their work is to extract parallelism. More importantly though, 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. More recently, translation-validation [31] compilers have used automatic theorem provers to validate compiler translations [21].

IX. Discussion and Future Work

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

We believe that the ACCORD annotations for programs that we would like programmers to write are also useful for reasons other than race-freedom. For instance, they form excellent formal documentations of the division of labor the algorithms employ, and can serve to better understand programs and maintain them. Programs written using the OpenMP [1] or CUDA [3] programming model generally have *deterministic* semantics, which is achieved by essentially ensuring non-interference in access to shared data among parallel threads. The sharing strategy is quite intricate due to performance considerations, specially at the boundaries. We believe ACCORD annotations can help prove non-interference, and hence determinism in such programs.

There is also promising 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 run-time 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 [29].

The annotations may help in sharing large data structure among concurrent agents or actors [5] efficiently i.e. without making copies of the data. For instance, Microsoft's agent language Axum [2], which currently uses a dynamically-enforced multiple-readers-single-writer protocol to control access to shared state by multiple actors, can benefit from the statically-enforced fine-grained sharing enabled by ACCORD annotations.

However, there are also several simpler thread contract languages that can be verified using static type-checking, and that are also useful in capturing sharing strategies (see [11] for an example for simple contracts based on nested regions). Also, simple static analysis algorithms that detect regions protected by locks are highly automatable and effective in practice. We believe a judicious combination of these various annotations with ACCORD would go far in building effective annotations for race-freedom.

Acknowledgements

The authors would like to acknowledge the feedback provided by Sarita Adve, Vikram Adve, Rob Bocchino, Darko Marinov during the course of this work. We would like to thank Rustan Leino and Shaz Qadeer for their guidance with using the Boogie/Spec# framework.

References

[1] OpenMP application program interface 3.0. *http://www.openmp.org*, 2008.

[2] Microsoft's Axum programming language. *http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx*, 2008-10.

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

[4] 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.

[5] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, MA, USA, 1986.

[6] 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.

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

[8] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. volume 3362, pages 49–69. Springer, 2004.

[9] M. Barnett, B. yuh Evan Chang, R. Deline, B. Jacobs, and K. R. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, pages 364–387. Springer, 2006.

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

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

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

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

[14] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *ESEC/SIGSOFT FSE*. ACM, 2009.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th international conference on Software engineering*, page 241. ACM, 2006.

[21] Y. Hu, C. Barrett, B. Goldberg, and A. Pnueli. Validating more loop optimizations. *Electron. Notes Theor. Comput. Sci.*, 141(2):69–84, 2005.

[22] C. Jones. Specification and design of (parallel) programs. *Information processing*, 83(9):321332, 1983.

[23] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL '08*, pages 171–182. ACM New York, NY, USA, 2008.

[24] G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):38, 2006.

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

[26] 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.

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

[28] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

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

[30] 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.

[31] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS '98*, LNCS. Springer-Verlag, 1998.

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

[33] 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.

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

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

[36] 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.

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

[38] 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.

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

[40] 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.