UPC-SPIN: A Framework for the Model Checking of UPC Programs *

Ali Ebnenasir

Department of Computer Science
Michigan Technological University
Houghton MI 49931, USA
aebnenas@mtu.edu

Abstract

This paper presents a software framework for the model checking of the inter-thread synchronization functionalities of Unified Parallel C (UPC) programs. The proposed framework includes a front-end compiler that generates finite models of UPC programs in the modeling language of the SPIN model checker. The model generation is based on a set of abstraction rules that transform the UPC synchronization primitives to semantically-equivalent code snippets in SPIN's modeling language. The back-end includes SPIN that verifies the generated model. If the model checking succeeds, then the UPC program is correct with respect to properties of interest such as data race-freedom and/or deadlock-freedom. Otherwise, the back-end provides feedback as *sequences of UPC instructions that lead to a data race or a deadlock from initial states*, called *counterexamples*. Using the UPC-SPIN framework, we have detected design flaws in several real-world UPC applications, including a program simulating heat flow in metal rods, parallel bubble sort, parallel data collection, and an integer permutation program. More importantly, for the first time (to the best of our knowledge), we have mechanically verified data race-freedom and deadlock-freedom in a UPC implementation of the Conjugate Gradient (CG) kernel of the NAS Parallel Benchmarks (NPB). We believe that UPC-SPIN provides a valuable tool for developers towards increasing their confidence in the computational results generated by UPC applications.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms High Performance Computing, Verification

Keywords PGAS, UPC, Model Checking

1. Introduction

The dependability of High Performance Computing (HPC) software is of paramount importance as researchers and engineers use HPC in critical domains of application (e.g., weather simulations, bio-electromagnetic modeling of human body, etc.) where design flaws may mislead scientists' observations. As such, we need to increase the confidence of developers in the accuracy of computational results. One way to achieve this goal is to devise techniques and tools that facilitate the detection and correction of concurrency failures[1] such as data races, deadlocks and livelocks. Due to the inherent non-determinism of HPC applications, software testing methods often fail to uncover concurrency failures as it is practically expensive (if not impossible) to check all possible interleavings of threads of execution. An alternative method is *model checking* [4, 11, 18] where we generate *finite models* of programs that represent a specific behavioral aspect (e.g., inter-thread synchronization functionalities), and exhaustively verify all interleavings of the finite model with respect to a property of interest (e.g., data race/deadlock-freedom). This paper presents a novel framework (see Figure 1) for model extraction and model checking of the Partitioned Global Address Space (PGAS) applications developed in Unified Parallel C (UPC).

While many HPC applications are developed using the Message Passing Interface (MPI) [9], there are important science and engineering problems that can be solved more efficiently in a shared memory model in part because the pattern of data access by independent threads of execution is irregular (e.g., the weighted matching problem [3, 17, 23]). As such, while there are tools for the model checking of MPI applications [20, 22, 25], we would like to enable the model checking of PGAS applications. The PGAS memory model aims at simplifying programming and increasing performance by exploiting data locality in a shared address space.

This paper presents a framework, called UPC-SPIN (see Figure 1), for model extraction and model checking of UPC applications using the SPIN model checker [11], thereby facilitating/automating the debugging of concurrency failures. UPC is a variant of the C programming language that supports the Single Program Multiple Data (SPMD) computation model with the PGAS memory model. UPC has been publicly available for many years and so many HPC users have experience with it. The proposed framework (see Figure 1) requires programmers to *manually* specify abstraction rules for model extraction in a Look-Up Table (LUT). Such abstraction rules are property-dependent in that for the same program and different properties/requirements (e.g., data race-freedom, deadlock-freedom), we may need to specify different abstraction rules. The abstraction rules specify how relevant UPC constructs are captured in the modeling language of the SPIN model checker [11]. After creating a LUT, UPC-SPIN automatically extracts a finite model

* This work was sponsored by the NSF grant CCF-0950678.

PGAS 2011 October 15-18, Galveston Island, Texas

[1] In the context of dependable systems [1], *faults* are events that cause a system to reach an *error* state from where system executions may deviate from its specification; i.e., a *failure* may occur.

from the source code and model checks the model with respect to properties of interest. The abstraction LUTs should be kept synchronized with any changes made in the source code. Our experience shows that after creating the first version of an LUT, keeping it synchronized with the source code has a relatively low overhead.

The proposed framework includes two components (see Figure 1): a front-end compiler and a back-end model checker. The front-end, called UPC Model Extractor (UPC-ModEx), extends the ModEx model extractor of ANSI C programs [12–14] in order to support the UPC grammar. UPC-ModEx takes a UPC program along with a set of abstraction rules (specified as a LUT) and automatically generates a Promela model (see Figure 1).[2] Promela [10] is the modeling language of SPIN, which is an extension of C with additional keywords and abstract data types for modeling concurrent computing systems. We expect that the commonalities of UPC and Promela will simplify the transformation of UPC programs to Promela models and will decrease the loss of semantics in such transformations. We present a set of built-in abstraction rules for the most commonly-used UPC synchronization primitives. After generating a finite model in Promela, developers specify properties of interest (e.g., data race-freedom) in terms of either simple assertions or more sophisticated temporal logic [8] expressions. SPIN verifies whether all executions of the model from its initial states satisfy the specified properties. If the model fails to meet the properties, then UPC-SPIN generates a sequence of program instructions that could lead to the failure from the initial state (Figure 1).

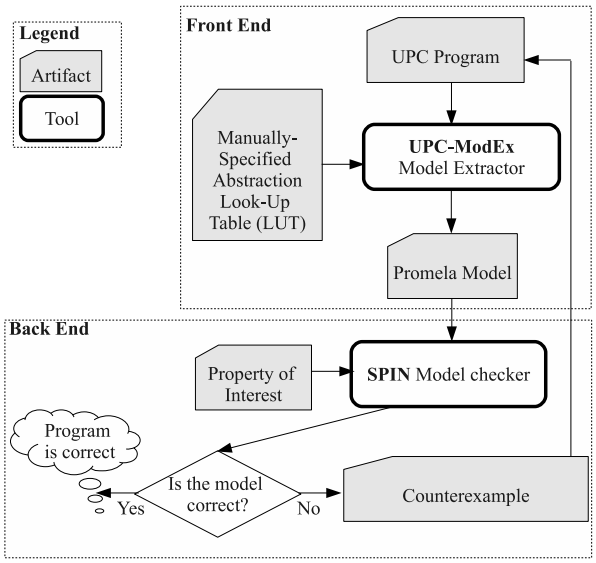

Figure 1. An overview of the UPC-SPIN framework.

We have used UPC-SPIN to detect and correct concurrency failures in small instances (i.e., programs with a few threads) of real-world UPC programs including parallel bubble sort, heat flow in metal rods, integer permutation and parallel data collection. More importantly, for the first time (to the best of our knowledge), we have generated a finite model of a UPC implementation of the Conjugate Gradient (CG) kernel of the NAS Parallel Benchmarks (NPB) [24]. We have model checked small instances of the extracted model of CG for data race-freedom and deadlock-freedom, thereby demonstrating its correctness. While the success of model checking means a model is correct, model checking can only be applied to small/moderate size models (see Section 6) due to the state space explosion problem. This may appear as a significant limitation considering the common belief amongst developers that some failures manifest themselves only when an application is scaled up (in terms of the number of processes and domain size of the input variables/parameters). Nonetheless, there is ample experimental evidence [16, 21] that most concurrency failures also exist in small instances of concurrent applications. Thus, a model checker that exhaustively verifies all possible interleavings can detect such failures in small instances of HPC applications.

[2] An executable copy of UPC-ModEx is available at `http://asd.cs.mtu.edu/projects/pgasver/index_files/upc-modex.html`.

Organization. Section 2 provides a background on UPC, model extraction and model checking. Section 3 discusses how the UPC-ModEx front-end extends ModEx to support the UPC constructs. Subsequently, Section 4 illustrates how we model check the Promela models of UPC programs with SPIN. Section 5 presents the model checking of an UPC implementation of the CG kernel of the NPB benchmarks [24]. Section 6 presents the time/space costs of model checking for our case studies. Finally, Section 7 makes concluding remarks and discusses future work.

2. Preliminaries

This section provides the basic concepts of UPC [5, 7] (Subsection 2.1), finite models of UPC programs (Subsection 2.2), an overview of model checking using SPIN [11] (Subsection 2.3) and concurrency failures and properties of interest (Subsection 2.4). Subsection 2.5 briefly discusses the internal working of the ANSI C Model Extractor (ModEx).

2.1 UPC: An Overview

UPC extends ANSI C by a SPMD model of computation where the same piece of code (e.g., Figure 2) is replicated in distinct threads of execution to process different data streams. The memory model of UPC (i.e., PGAS) divides its address space into shared and private parts. The shared area of the memory space is partitioned into THREADS sections, where THREADS is a system constant representing the total number of threads. Each thread has a private memory space and is also uniquely associated with a shared section, called its *affinity*; e.g., `A[MYTHREAD]` in Lines 10-11 of Figure 2, where `MYTHREAD` denotes thread's own thread number. To support parallel programming, UPC augments C with a set of synchronization primitives, a work-sharing iteration statement `upc_forall` and a set of collective operations. Figure 2 demonstrates an integer permutation application that takes an array of distinct integers (see array A in Line 2 of Figure 2) and randomly generates a permutation of A without creating any duplicate/missing values. Shared data structures are explicitly declared with a `shared` type modifier. A shared array of `THREADS` locks (of type `upc_lock_t`) is declared in Line 3. Each thread initializes `A[MYTHREAD]` (Lines 10-11) and randomly chooses an array element (Line 14) to swap with the contents of `A[MYTHREAD]`.

2.2 Finite Models of UPC Programs

Let p be a UPC program with a fixed number of threads, denoted THREADS > 1. A model of p is a non-deterministic finite state transition system denoted by a triple $\langle V_p, \delta_p, I_p \rangle$ representing the inter-thread synchronization functionalities of p, called the *synchronization skeleton* of p. V_p represents a finite set of synchronization variables with finite domains. A *synchronization variable* is a shared variable (e.g., locks) between multiple threads used for synchronizing access to shared resources/variables. A *control variable* (e.g., program counter) captures the execution control of a thread. A *state* is a unique valuation of synchronization and control variables. An ordered pair of states (s_0, s_1) denotes a transition. A *thread* contains a set of transitions, and δ_p denotes the union of the set of transitions of threads of p. We use actions (a.k.a *guarded commands*) to represent sets of program transitions. An action is of the form $grd \rightarrow stmt$, where the guard grd is an expression in terms of model variables and the statement $stmt$ updates model

variables. When the guard grd holds (i.e., the action is *enabled*), the statement $stmt$ can be executed, which accordingly updates some variables. Each action captures a set of transitions of a specific thread. I_p represents a set of initial states. The state space of p, denoted $\mathcal{S}_p$, is equal to the set of all states of p. A *state predicate* is a subset of $\mathcal{S}_p$; i.e., defines a function from $\mathcal{S}_p$ to {true, false}. A state predicate $\mathcal{X}$ is true (i.e., holds) in a state s iff (if and only if) $s \in \mathcal{X}$. A *computation* (i.e., *synchronization trace*) of p is a *maximal* sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states s_i, where $s_0 \in I_p$ and each transition (s_i, s_{i+1}) belongs to an action of some thread; i.e., $(s_i, s_{i+1}) \in \delta_p$ for $i \geq 0$. That is, either σ is infinite, or if σ is a finite sequence $\langle s_0, s_1, \cdots, s_f \rangle$, then no thread is enabled at s_f, where an *enabled thread* has at least one enabled action.

2.3 Model Checking, SPIN and Promela

Explicit-state model checkers (e.g., SPIN [11]) create models as finite-state machines represented as directed graphs in memory, where each node captures a unique state of the model and each arc represents a state transition. Symbolic model checkers create models as Binary Decision Diagrams (BDDs) (e.g., SMV [18]) and are mostly used for hardware verification. If model checking succeeds, then the model is correct. Otherwise, model checkers provide scenarios as to how an error is reached from initial states, called *counterexamples*. SPIN is a explicit-state model checker with a C-like modeling language, called Promela [10]. A Promela model comprises (1) a set of variables, (2) a set of (concurrent) processes modeled by a predefined type, called *proctype*, and (3) a set of asynchronous and synchronous channels for inter-process communications. The semantics of Promela is based on an operational model that defines how the actions of processes are interleaved. Actions can be atomic or non-atomic, where an atomic action (denoted by atomic {} blocks in Promela) ensures that the guard evaluation and the execution of the statement is uninterrupted.

2.4 Concurrency Failures and Properties of Interest

To verify a model using model checkers, developers have to specify safety and liveness properties of interest. Intuitively, a *safety* property stipulates that nothing bad ever happens in any computation. Data race-freedom and deadlock-freedom are instances of safety properties. A *data race* occurs when multiple threads access shared data simultaneously, and at least one of those accesses is a write [19]. A block of statements accessing shared data is called a *critical section* of the code, denoted CS_i for thread $0 \leq i <$THREADS; e.g., Lines 19-21 and 29-31 in Figure 2 where threads perform the swapping. A data race could occur when two or more threads are in their critical sections. However, the upc_lock statements in Lines 17-18 and 27-28 ensure that each thread gets exclusive access to its critical section so no data races occur. The section of the code where a thread tries to enter its critical section is called its *trying section*, denoted TS_i for thread i (e.g., Lines 17-18 and 27-28). A program is *deadlocked* when no thread can *make progress* in entering its critical section. Deadlocks occur often due to circular-wait scenarios when a set of threads $T_1, \cdots, T_k$ wait for one another in a circular fashion (e.g., T_1 waits for T_2, T_2 waits for T_3 and so on until T_k which waits for T_1). Formally, a deadlock state has no outgoing transitions. The two if-statements in Lines 16 and 26 of Figure 2 impose a total order on the way lock variables are acquired in order to break circular waits.

In the UPC program of Figure 2, a safety property stipulates that *it is always the case* that no two threads have access to the same array cell. In SPIN, such properties are formally specified using the *always* operator in Linear Temporal Logic (LTL) [8], denoted $\Box$. The example UPC code of Figure 2 ensures that the safety property $\Box(((\text{main}[i] : s = j) \vee (\text{main}[j] : s = i)) \Rightarrow \neg(CS_i \wedge CS_j))$ is met by acquiring locks ($0 \leq i, j <$ THREADS), where CS_i is a state predicate representing that thread i is in its critical section (i.e., Lines 19-21 or Lines 29-31) and '$\text{main}[i] : s$' denotes the value of the local variable s in thread i created from the proctype 'main' in Figure 4.

A *progress* property states that it is *always* the case that if a predicate P becomes true, then another predicate Q will *eventually* hold. We denote such progress properties by $\mathcal{P} \rightsquigarrow \mathcal{Q}$ (read it as 'P leads to Q') [8]. For example, in the example UPC program of Figure 2, we specify progress for each thread i ($0 \leq i <$ THREADS) as $TS_i \rightsquigarrow CS_i$; i.e., it is always the case that if thread i is in its trying section (represented by the predicate TS_i), then it will eventually enter its critical section (i.e., CS_i holds).

```
1  // Declaring shared global variables
2  shared int A[THREADS];
3  upc_lock_t *shared lk[THREADS];
4
5  int main(int argc, char **argv) {
6    int i, s, temp;
7
8      // The body of each thread starts
9      // Initialize the array A with distinct integers
10     i = MYTHREAD;
11     A[i] = i;
12     upc_barrier;
13    // Randomly generate a swap index
14    s = (int)lrand48() % (THREADS);
15
16    if (s<i) {
17         upc_lock(lk[i]);  // Acquire locks
18         upc_lock(lk[s]);
19           temp = A[i];  // Swap
20           A[i] = A[s];
21           A[s] = temp;
22         upc_unlock(lk[s]);  // Release locks
23         upc_unlock(lk[i]);
24     }
25
26    if (i<s) {
27         upc_lock(lk[s]);  // Acquire locks
28         upc_lock(lk[i]);
29           temp = A[i];  // Swap
30           A[i] = A[s];
31           A[s] = temp;
32         upc_unlock(lk[i]);  // Release locks
33         upc_unlock(lk[s]);
34     }
35 }
```

Figure 2. Excerpts of the integer permutation program in UPC.

2.5 ModEx: Model Extractor of ANSI C Programs

Since in Section 3 we extend the front-end compiler of the ANSI C Model Extractor (ModEx) [12–14] to support the UPC grammar, this section presents an overview of ModEx, which is a software tool for extracting finite models from ANSI C programs.

ModEx generates finite models of C programs in three phases, namely *parsing*, *interpretation using abstraction rules* and *optimization for verification*. In the **parsing phase**, ModEx generates an uninterpreted parse tree of the input source code that captures the control flow structure of the source code and the type and scope of each data object. All basic linguistic constructs of C (e.g., declarations, assignments, conditions, function calls, control statements) are collected in the parse tree and remain uninterpreted. The parse tree also keeps some information useful for representing the results of model checking back to the level of source code (e.g., association between the lines of source code and the lines of code in the model). The essence of the **interpretation phase** is based on a *tabled-abstraction* method that pairs each parse tree construct with an interpretation in the target modeling language. ModEx can perform such interpretation based on either a default set of abstraction rules or programmer-defined abstraction rules. Different

types of abstractions can be applied to the nodes of the parse tree including *local slicing* and *predicate abstraction*. In local slicing, data objects that are irrelevant to the property of interest (e.g., local variables that have no impact on inter-thread synchronizations) are sliced away. Any operation (e.g., assignments, function calls) performed on or dependent upon irrelevant data objects are sliced away and replaced with a null operation in the model. In predicate abstraction, if there are variables in the source code whose domains include more information than necessary for model checking, then they can be abstracted as Boolean variables in the model. For example, consider a variable $0 \leq$temp ≤ 100 that stores the temperature of a boiler tank (in Celsius), and the program should turn off a burner if the temperature is 95 degrees or above. For verifying whether the burner is off when temp ≥ 95, a Boolean variable can capture the value of a predicate representing whether or not temp is below 95. In the **optimization** phase, ModEx uses a set of rewrite rules to simplify some generated statements in Promela and to eliminate statements that have no impact on verification. For example, the guarded command $false \rightarrow x = 0$ in Promela can be omitted without any impact on the result of model checking because the guard is always *false* and the action is never enabled/executed.

3. UPC Model Extractor (UPC-ModEx)

This section discusses how we extend ModEx to support the parsing (Section 3.1) and the interpretation (Section 3.2) of UPC constructs in UPC-ModEx. Section 3.3 discusses how we abstract read/write accesses to shared data, and Section 3.4 demonstrates model extraction in the context of the integer permutation program in Figure 2.

3.1 Parsing UPC Constructs

The ANSI C ModEx lacks support for the UPC extension of C including type qualifiers, unary expressions, iteration statements, synchronization statements and UPC collectives. Due to space constraints, we omit the extension for unary expressions (see [6] for details). The extension for UPC collectives is outside the scope of this paper.

Type qualifiers. UPC includes three type qualifiers, namely shared, strict and relaxed. The shared type qualifier is used to declare data objects in the shared address space. We augment the grammar using the following rules in the BNF form [2]:

- type_qual: CONST | VOLATILE | shared_type_qual | reference_type_qual
- shared_type_qual: "shared" | "shared" '[' opt_const_expr ']' | "shared" '[*]'

The reference type qualifiers strict and relaxed are used to declare variables that are accessed based on the strict or relaxed memory consistency model.

- reference_type_qual: "relaxed" | "strict"

We note that, in this paper, we focus on model checking in the strict consistency model.

Iteration statements. In addition to regular iteration statements of C, UPC has a work-sharing iteration statement, denoted upc_forall. The upc_forall statement enables programmers to distribute independent iterations of a for-loop across distinct threads. The grammar of upc_forall in BNF is as follows:

- forall_stemnt: "upc_forall" '(' opt_expr ';'opt_expr ';' opt_expr ';' affinity_expr ')' stemnt
- affinity_expr: "continue" | opt_expr

The affinity expression affinity_expr determines which thread executes which iteration of the loop depending on the affinity of the data objects referred in affinity_expr. If affinity_expr is an integer expression $expr$, then each thread executes the body of the loop when MYTHREAD is equal to ($expr \ mod$ THREADS). If affinity_expr is continue or not specified, then each thread executes every iteration of the loop body.

Synchronization statements. The most commonly used synchronization statements in UPC include upc_barrier, upc_wait and upc_notify statements. Moreover, UPC has a new type upc_lock_t that enables programmers to declare lock variables for synchronizing access to shared resources/data. The two functions upc_lock() and upc_unlock() are used to acquire and release shared variables of type upc_lock_t. The grammar of the synchronization statements is as follows:

- upc_barrier_stemnt: "upc_barrier" opt_expr ';'
- upc_wait_stemnt: "upc_wait" opt_expr ';'
- upc_notify_stemnt: "upc_notify" opt_expr ';'

We extend ModEx to support the compilation of UPC-specific constructs discussed above.

3.2 Interpreting UPC Constructs Using Abstraction

This section presents a set of abstraction rules that we have developed for model extraction from UPC programs. We use ModEx commands [12–14] for the specification of such rules. Each rule is of the form:

left-hand side right-hand side

The left-hand side is a UPC statement and the right-hand side could be either a piece of Promela code that should be generated corresponding to the left-hand side or an abstraction command that specifies how the left-hand side should be treated in the model (see Table 1 for some example commands). The LUT comprises a set of abstraction rules. For example, the *skip* command generates a null operation in the Promela model corresponding to the left-hand side, the *hide* command conceals the left-hand side in the model; i.e., nothing is generated, and the *keep* command preserves the left-hand side in the Promela model. Some abstraction commands enable string matching and replacement, such as the *Substitute* command. For example, the command

"Substitute MYTHREAD _pid"

replaces any occurrence of MYTHREAD in the UPC code with _pid in the model, where _pid captures a unique identifier for each proctype in Promela. There is also an *Import* command that includes the data objects name from the UPC source code inside the Promela model with the global scope or the scope of a specific proctype in Promela.

Table 1. Sample Abstraction Commands

Command	Meaning
skip	Replace with a null operation
hide	Conceal in the model
keep	Preserve in the model
Substitute P_1 P_2	Substitute any occurrence of P_1 with P_2
Import name scope	Include name with a scope of 'scope'

We present the following abstraction rules for model generation from UPC programs (see [6] for more rules):

Rule 1: upc_lock() The upc_lock(upc_lock_t *lk) function locks a *shared* variable of type upc_lock_t. If the lock is already acquired by some thread, the calling thread waits for the lock to be released. Otherwise, the calling thread acquires the lock *lk atomically. The corresponding Promela code is as follows:

```
1 bool lk;  // Global lock variable
2 atomic{ !lk -> lk=true; }
```

Line 2 represents an atomic guarded command in Promela that sets the lock variable lk to true (i.e., acquires lk) if lk is available. Otherwise, the atomic guarded command is blocked.

Rule 2: upc_unlock() The upc_unlock(upc_lock_t *lk) is translated to an assignment lk = false in Promela. Assignments are executed atomically in Promela.

Rule 3: upc_notify We use two global integer variables barr and proc to implement the semantics of upc_notify in Promela. Initially, the value of barr is equal to THREADS. To demonstrate that it has reached a notify statement, each thread *atomically* decrements the value of barr and sets the flag proc to zero. Notice that barr and proc are updated atomically because they are shared variables in the model and a non-atomic update may cause data races.

```
1 atomic{ barr = barr -1;  proc=0;}
```

Rule 4: upc_wait Once reached a upc_wait statement, a thread waits until the value of barr becomes zero; i.e., all threads have reached their notify statement in the current synchronization phase. The value of proc is set to 1 indicating that some thread has observed that barr has become zero. Afterwards, each thread increments barr and waits until all threads increment barr or some thread has witnessed that barr has become equal to THREADS in the current phase (i.e., proc has been set to 0).

```
1 (barr == 0) || (proc == 1) -> proc = 1;
2  barr = barr + 1;
3 (barr == THREADS) || (proc == 0) -> proc = 0;
```

Rule 5: (Split-Phase) upc_barrier The upc_barrier is in fact the union of a pair of upc_notify and upc_wait statements. Separate use of upc_notify and upc_wait implements the split-phase barrier synchronization. Split-phase barrier can reduce the busy-waiting overhead of barrier synchronizations by allowing each thread to perform some local computations between the time it reaches a notify statement and the time it reaches a wait statement.

Rule 6: upc_forall To model the work-sharing iteration statement upc_forall in Promela, we first explain how regular for-loops in C are modeled by ModEx. Then, we describe how we extract Promela models from upc_forall statements. Consider a C for-loop "for (init; cond; cntr_update){ stmtBlk;}", where init denotes the initialization of the loop counter, the cond represents the termination condition, cnt_update updates the loop counter and stmtBlk is the statement block in the loop body. The following Promela statements model such a C for-loop.

```
1 init;
2 do
3   :: (cond) -> {stmtBlk; update_cntr;}
4   :: else -> break;
5 od;
```

Line 1 initializes the loop counter and the do-od loop captures the control logic of the for-loop. That is, if the condition cond holds, then the loop body is executed and the loop counter is updated. Otherwise, the do-od loop breaks out. The upc_forall statement distributes the loop iterations amongst the threads based on an affinity expression "for (init; cond; cntr_update; affinity_expr){ stmtBlk;}". If affinity_expr is equal to 'continue' or is unspecified, then the model generated for upc_forall is similar to a regular C for-loop. Otherwise, affinity_expr could be either an integer expression intExpr or a reference to the k-th element of a shared array. For the first case, we have the following piece of code instead of Line 3 of the above do-od loop.

```
:: (cond) -> {if (_pid == intExpr){ stmtBlk; }
                                    update_cntr;}
```

For the second case, consider a shared array declared as "shared [block-size] array A[number-of-elements]". Since UPC distributes the elements of the array A in the shared memory space in a round robin fashion where at least block-size elements are associated with the affinity of each thread, we generate the following Promela code when the k-th element of the array A is referenced in affinity_expr:

```
:: (cond) -> {if (_pid == ((k/block-size)%THREADS)){
                          stmtBlk; }  update_cntr;}
```

3.3 Abstracting Shared Data Accesses

In the model checking of concurrent programs for data race-freedom, the objective is to check whether or not multiple threads have simultaneous access to shared data where at least one thread performs a write operation. Thus, the contents of shared variables and the way it is accessed (i.e., via pointers or by name) are irrelevant to verification; rather it is the type of read/write operation on the shared data that should be captured in a model. For this reason, corresponding to each shared variable x, we consider two bits in the Promela model; one represents whether a read operation is being performed on x and the other captures the fact that x is being written. Accordingly, if a shared array is used in the UPC program, its corresponding model will include two bit-arrays. For example, corresponding to the array A in Figure 2, we consider the following bit arrays in its Promela model:

```
1 bit read_A[THREADS];
2 bit write_A[THREADS];
```

The bit read_A$[i]$ (for $0 \leq i \leq$ THREADS-1) is 1 if and only if a read operation is performed on A$[i]$. Likewise, write_A$[i]$ is 1 if and only if A$[i]$ is written. Thus, corresponding to any read (respectively, write) operation on A$[i]$ in the UPC code, we set read_A$[i]$ (respectively, write_A$[i]$) to 1 in the Promela model.

3.4 Example: Promela Model of Integer Permutation

For model extraction, UPC-ModEx needs two input files: the input UPC program and a text file that contains the abstraction LUT. Figure 3 illustrates the LUT for the program in Figure 2:

```
1  %F Locks.c
2  %X -L main.lut
3  %L
4  Import                 i          main
5  Import                 s          main
6  Substitute             MYTHREAD   _pid
7  A[i] = i       hide
8  upc_barrier    atomic{ barr = barr -1;  proc=0;}
9                 (barr == 0) || (proc == 1) -> proc = 1;
10                barr = barr + 1  ;
11                (barr == THREADS)||(proc == 0)-> proc = 0;
12 s=((int )lrand48()...  select(s: 0 .. THREADS-1)
13 upc_lock(lk[i])        atomic{ !lk[i] -> lk[i] = true }
14 upc_lock(lk[s])        atomic{ !lk[s] -> lk[s] = true }
15 upc_unlock(lk[i])      lk[i] = false
16 upc_unlock(lk[s])      lk[s] = false
17 t=A[i]                 read_A[i]=1;
18                        read_A[i]=0;
19 A[i]=A[s]               read_A[s]=1;
20                        read_A[s]=0;
21                        write_A[i]=1;
22                        write_A[i]=0;
23 A[s]=t                 write_A[s]=1;
24                        write_A[s]=0;
```

Figure 3. The abstraction file for the program in Figure 2, where THREADS $= 4$.

While the commands used in this file are taken from ModEx, the abstraction rules that specify how a model is generated from the UPC program are our contributions. The first line in Figure 3 (i.e., command %F) specifies the name of the source file from which we want to extract a model. Line 2 (i.e., command %X) expresses that UPC-ModEx should extract a model of the main function using the subsequent abstraction rules. Line 3 (i.e., command %L) denotes the start of the look-up table that is used for model extraction. Lines 4 and 5 define that the variables i and s should be included as local variables in the proctype that is generated corresponding to the main function of the source code. Since the contents of array A is irrelevant to the verification of data race/deadlock-freedom, we hide the statement A$[i] = i$ in the model, where i is set to MYTHREAD. We apply Rule 5 (presented in Section 3.2) for the abstraction of

upc_barrier (Lines 8-11 in Figure 3). Line 14 of Figure 2 (i.e., s = (int)lrand48() % (THREADS)) assigns a randomly-selected integer (between 0 and THREADS−1) to variable s. The semantics of Line 14 is captured by a 'select(v : $L..H$)' statement in Promela (e.g., Line 12 in Figure 3), where a random number between L and H (inclusive) is assigned to variable v. The value of the variable s determines the array cell with which the value of $A[i]$ should be swapped by thread i. Lines 13-16 include the rules for the abstraction of upc_lock() and upc_unlock() functions. Lines 17-24 illustrate the rules used to abstract read/write accesses to shared data (as explained in Section 3.3). For example, the assignment $A[i] = A[s]$ in UPC is translated to four assignments demonstrating how $A[s]$ is read and $A[i]$ is written.

Taking the program in Figure 2 and the abstraction file of Figure 3, UPC-ModEx generates the Promela model in Figure 4. Lines 1-6 have been added manually. Line 1 defines a macro that captures the system constant THREADS; in this case 4 threads. Lines 2-6 declare global shared variables that are accessed by all proctypes in the model. The prefix active in Line 8 means that a set of processes are declared that are *active* (i.e., running) in the initial state of the model. The suffix [THREADS] specifies the number of instances of the main proctype that are created by SPIN. Lines 11-14 implement upc_barrier. Each proctype randomly assigns a value between 0 and THREADS−1 to variable s (Line 17) and then performs the swapping in either one of the if-statements in Lines 18 or 34. The automatically-generated line numbers that are written as comments associate the instructions in the UPC source code with the statements in the model.

4. Model Checking with SPIN

In order to verify a model with respect to a property, we first have to specify the property in terms of the data flow or the control flow of the model (or both). For example, to verify the model of Figure 4 for lack of simultaneous read and write operations, we first determine the conditions under which a shared datum is read and written by multiple threads at the same time. (Section 5 illustrates a case where we verify freedom from simultaneous writes.) Using the abstractions defined for shared data accesses in Section 3.3, we define the following macro for the Promela model of Figure 4:

```
1 #define race_0  (read_A[0] && write_A[0])
```

The macro race_0 defines conditions under which the array cell A[0] is read and written at the same time; i.e., there is a race condition on A[0]. Likewise, we define macros representing race conditions for other cells of array A. To express data race-freedom in SPIN, we specify the temporal logic expression □ !race_0 meaning that *it is always the case that the condition race_0 is false*. The data race-freedom in this case is guaranteed by the use of upc_lock statements in Lines 17-18 and 27-28 in Figure 2, which are translated as Lines 19-20 and 35-36 in Figure 4. Nonetheless, the use of locks often causes deadlocks in concurrent programs due to circular waiting of threads for shared locks. To verify deadlock-freedom, we must make sure that each thread eventually terminates; i.e., eventually reaches Line 50 of Figure 4. To specify this property, we define the following macros:

```
1 #define fin_0  (main[0]@P)
```

The macro fin_0 is defined in terms of the control flow of the first instance of the main proctype, denoted main[0], which means that thread 0 is at the label P (inserted in Line 50). Thus, if thread 0 eventually reaches its last statement (which is a null operation in Promela denoted by skip), then it is definitely not deadlocked. Such a property is specified as the temporal logic expression ◊ fin_0, where ◊ denotes the eventuality operator in temporal logic. SPIN verifies the integer permutation program with respect to data race-freedom and deadlock-freedom properties.

```
1  #define THREADS 4
2  int barr = THREADS;
3  int proc = 0;
4  bool lk[THREADS];
5  bit read_A[THREADS];
6  bit write_A[THREADS];
7
8  active [THREADS] proctype main() {
9   int i, s;
10  i=_pid;   /* line  42 */
11  atomic{ barr = barr -1;  proc=0; } /* line  39 */
12  (barr == 0) || (proc == 1) -> proc = 1;
13  barr = barr + 1;
14  (barr == THREADS) || (proc == 0) -> proc = 0;
15                /* line  44 */
16
17  select(s: 0 .. THREADS-1);
18  if ::  (s<i) -> {          /* line  54 */
19       atomic{ !lk[i] -> lk[i] = 1 }; /* line  59 */
20       atomic{ !lk[s] -> lk[s] = 1 }; /* line  60 */
21       read_A[i]=1;          /* line  62 */
22       read_A[i]=0;
23       read_A[s]=1;          /* line  63 */
24       read_A[s]=0;
25       write_A[i]=1;
26       write_A[i]=0;
27       write_A[s]=1;           /* line  64 */
28       write_A[s]=0;
29       lk[i] = 0;          /* line  66 */
30       lk[s] = 0;          /* line  67 */
31       }
32     :: else;          /* line  67 */
33  fi;
34  if ::  (s>i) -> {          /* line  67 */
35       atomic{ !lk[s] -> lk[s] = 1 }; /* line  70 */
36       atomic{ !lk[i] -> lk[i] = 1 }; /* line  71 */
37       read_A[i]=1;          /* line  73 */
38       read_A[i]=0;
39       read_A[s]=1;          /* line  74 */
40       read_A[s]=0;
41       write_A[i]=1;
42       write_A[i]=0;
43       write_A[s]=1;          /* line  75 */
44       write_A[s]=0;
45       lk[i] = 0;          /* line  77 */
46       lk[s] = 0;          /* line  78 */
47       }
48     :: else;          /* line  78 */
49  fi;
50  P: skip;  }
```

Figure 4. The Promela model generated for the program in Figure 2.

4.1 Example: Heat Flow

The Heat Flow (HF) program includes THREADS>1 threads and a shared array t of size THREADS×$regLen$, where $regLen > 1$ is the length of a region vector accessible to each thread. That is, each thread i ($0 \leq i \leq$ THREADS−1) has read/write access to array cells $t[i * regLen]$ up to $t[((i + 1) * regLen) - 1]$. The shared array t captures the transfer of heat in a metal rod and the HF program models the heat flow in the rod. Figure 5 presents an excerpt of the UPC code of HF.

Each thread performs some local computations and then all threads synchronize with the upc_barrier in Line 5. The base of the region of each thread is computed by MYTHREAD∗ regLen in Line 6. Each thread continuously executes the code in Lines 7 to 26. In Lines 8-11, the local value of tmp[0] is initialized. Then, in Lines 12 to 16, each thread i, where $0 \leq i \leq$ THREADS−1, first computes the heat intensity of the cells $t[base]$ to $t[base + regLen - 3]$ in its own region. Subsequently, every thread, except the last one, updates the heat intensity of $t[base + regLen - 1]$ (see Lines 17-

```
1  shared double t[regLen*THREADS];
2  double tmp[2];
3  double e, etmp;
4  . . .  // Perform some local computations
5  upc_barrier;
6  base = MYTHREAD*regLen;
7  for (j =0; j < regLen+1; j++) {
8   if (MYTHREAD == 0) { tmp[0] = t[0]; }
9   else {
10     tmp[0] = (t[base-1] + t[base] + t[base+1])/3.0;
11     e = fabs(t[base]-tmp[0]); }
12  for (i=base+1; i<base+regLen-1; ++i) {
13    tmp[1] = (t[i-1] + t[i] + t[i+1]) / 3.0;
14    etmp = fabs(t[i]-tmp[1]);
15    t[i-1] =  tmp[0];
16    }
17  if (MYTHREAD < THREADS-1) {
18    tmp[1] = (t[base+regLen-2] +
19                  t[base+regLen-1] +
20                         t[base+regLen]) / 3.0;
21    etmp = fabs(t[base+regLen-1]-tmp[1]);
22    t[base+regLen-1] = tmp[1];
23    }
24  upc_barrier;
25  t[base+regLen-2] = tmp[0];
26 }
```

Figure 5. Excerpt of the Heat Flow (HF) program in UPC.

23). Before updating $t[base + regLen - 2]$ in Line 25, all threads synchronize using upc_barrier. Our objective is to verify whether or not there are any simultaneous read-write operations in HF. Since no thread writes in another thread's region, no simultaneous writes occur. The significance of this example is that the access to shared data is changed dynamically as each thread updates the value of heat flow. Moreover, despite the small number of lines of code in this example, it is difficult to *manually* identify where the data races may occur.

Abstraction Look-Up Table (LUT) for HF. We present the abstraction LUT of the HF program below. Lines 1-7 of the table include the local data and simple mapping rules. The rest of the abstraction table includes 11 entries located in Lines 8, 10, 14, 21, 24, 31, 33, 35, 38, 40 and 42. Each entry includes a left-hand side and a right-hand side defined based on the rules presented in Sections 3.2 and 3.3. Hence, we omit the explanation of the abstraction rules of the HF program. Notice that the arrays read_t and write_t have been declared for the abstraction of data accesses to the shared array t (as explained in Section 3.3).

```
1  %F fwup.c
2  %X -L main.lut
3  %L
4  Import   i   main
5  Import   j   main
6  Import   base   main
7  Substitute   MYTHREAD   _pid
8  tmp[0]=t[0]    read_t[0]=1;
9                 read_t[0]=0;
10 upc_barrier    atomic barr = barr -1;  proc=0;
11                (barr == 0) || (proc == 1) -> proc = 1;
12                barr = barr + 1  ;
13                (barr==THREADS)||(proc==0) -> proc = 0;
14 tmp[0]=(((t[(base-1)]+t[base])+t[(base+1)])/3)
15                read_t[base-1] = 1;
16                read_t[base-1] = 0;
17                read_t[base] = 1;
18                read_t[base] = 0;
19                read_t[base+1] = 1;
20                read_t[base+1] = 0;
21 e=fabs((t[base]-tmp[0]))    read_t[base]=1;
22                             read_t[base]=0;
23
24 tmp[1]=(((t[(i-1)]+t[i])+
25                t[(i+1)])/3) read_t[i-1] = 1;
26                             read_t[i-1] = 0;
27                             read_t[i] = 1;
28                             read_t[i] = 0;
29                             read_t[i+1] = 1;
30                             read_t[i+1] = 0;
31 etmp=fabs((t[i]-tmp[1]))    read_t[base]=1;
32                             read_t[base]=0;
33 t[(i-1)]=tmp[0]             write_t[i-1] = 1;
34                             write_t[i-1] = 0;
35 etmp=fabs((t[((base+regLen)-1)]-tmp[1]))
36                read_t[((base+regLen)-1)]=1;
37                read_t[((base+regLen)-1)]=0;
38 t[((base+regLen)-1)]=tmp[1] write_t[((base+regLen)-1)]=1;
39                             write_t[((base+regLen)-1)]=0;
40 t[((base+regLen)-2)]=tmp[0] write_t[base+regLen-2]=1;
41                             write_t[base+regLen-2]=0;
42 tmp[1]=(((t[((base+regLen)-2)]+ t[((base+regLen)-1)])
43 +t[(base+regLen)])/3) read_t[((base+regLen)-2)] = 1;
44                       read_t[((base+regLen)-2)] = 0;
45                       read_t[((base+regLen)-1)] = 1;
46                       read_t[((base+regLen)-1)] = 0;
47                       read_t[(base+regLen)] = 1;
48                       read_t[(base+regLen)] = 0;
```

The Promela model of HF. UPC-ModEx generates the following Promela model for the HF program using its abstraction LUT. This is an instance with 3 threads and region size of 3 for each thread. The SPIN model checker creates THREADS instances of the main proctype as declared in Line 8. We omit the explanation of the Promela model of HF as it has been generated with the rules defined in Sections 3.2 and 3.3.

```
1  #define regLen 3
2  #define THREADS 3
3  int barr = THREADS;
4  int proc = 0;
5  bit read_t[regLen * THREADS];
6  bit write_t[regLen * THREADS];
7
8  active [THREADS] proctype main() {
9  int base ;        /* mapped */
10 int i ;        /* mapped */
11 int j ;        /* mapped */
12 atomic{ barr = barr -1;  proc=0; }   /* line  50 */
13 (barr == 0) || (proc == 1) -> proc = 1;
14 barr = barr + 1 ;
15 (barr == THREADS) || (proc == 0) -> proc = 0;
16 base = _pid * regLen;  /*  line  55 */
17 j=0;          /* line  60 */
18L_0: do
19    :: (j<(regLen +1)) -> { /* line  60 */
20       if :: (_pid==0) /* line  60 */
21             read_t[0]=1; read_t[0]=0; /* line  64 */
22          :: else;         /* line  64 */
23       read_t[base-1] = 1; read_t[base-1] = 0;
24       read_t[base] = 1; read_t[base] = 0;
25       read_t[base+1] = 1; read_t[base+1] = 0;
26       /* line  68 */
27       read_t[base]=1; read_t[base]=0; /* line  69 */
28       fi;
29       i = base +1;  /* line  71 */
30L_1: do
31     :: (i < base+regLen-1) -> {
32        read_t[i-1] = 1;  read_t[i-1] = 0;
33        read_t[i] = 1; read_t[i] = 0;
34        read_t[i+1] = 1; read_t[i+1] = 0;
35        /* line  73 */
36        read_t[base]=1; read_t[base]=0; /* line  74 */
37        write_t[i-1] = 1; write_t[i-1] = 0; /* line  76 */
38        i = i +1;  /* line  76 */
39        }
```

```
40      :: else  -> break           /* line  76 */
41      od;
42      if :: (_pid<(THREADS-1))  /* line  76 */
43         read_t[((base+regLen)-2)] = 1; /* line  81 */
44         read_t[((base+regLen)-2)] = 0;
45         read_t[((base+regLen)-1)] = 1;
46         read_t[((base+regLen)-1)] = 0;
47         read_t[(base+regLen)] = 1;
48         read_t[(base+regLen)] = 0;
49         read_t[((base+regLen)-1)]=1; /* line  83 */
50         read_t[((base+regLen)-1)]=0;
51         write_t[((base+regLen)-1)]=1; /* line  87 */
52         write_t[((base+regLen)-1)]=0;
53         :: else;         /* line  87 */
54      fi;
55      atomic{barr = barr -1; proc=0;} /* line  89 */
56      (barr == 0) || (proc == 1) -> proc = 1;
57      barr = barr + 1 ;
58      (barr == THREADS) || (proc == 0) -> proc = 0;
59      write_t[base+regLen-2]=1;
60      write_t[base+regLen-2]=0; /* line  91 */
61      j = j +1 ; /* line  91 */
62     }
63    :: else -> break  /* line  91 */
64  od;
65}
```

Model checking for data race-freedom. We specify data race conditions where at least a write operation is performed simultaneously with multiple reads as the following macros ($0 \leq i \leq$ THREADS$*$regLen-1):

```
1 #define race_i (read_t[i] && write_t[i])
```

To verify data race-freedom, we use SPIN to check whether the following property is correct: *it is always the case that there are no simultaneous reads and writes on array cell* $t[i]$. This parameterized property is formally expressed as $\Box\neg$ race_i. We verify the model of the HF program for data race-freedom on all memory cells of array t; i.e., $0 \leq i \leq$ THREADS$*$regLen-1. For an instance of the model where THREADS$=3$ and regLen$=3$, SPIN finds data races on the boundary array cells. That is, there are data races on $t[2], t[3], t[5]$ and $t[6]$.

Counterexamples. SPIN returns a counterexample, where a read in Line 23 and a write in Line 51 of the above Promela model occur simultaneously. Specifically, this counterexample demonstrates that while Thread 0 is writing $t[2]$ in Line 51, Thread 1 could be reading $t[2]$ in Line 23. The translation of this counterexample in the source code is that a data race occurs on $t[2]$ when thread 0 is writing $t[2]$ in Line 22 of Figure 5 and thread 1 is reading $t[2]$ in Line 10 of Figure 5. Another data race occurs when thread i is writing its t[base] in Line 15 of Figure 5 in the first iteration of the for-loop of Line 12, and thread $i-1$ is reading t[base+regLen] in Line 20, for $1 \leq i \leq$ THREADS-1. This example illustrates how model checking could simplify the detection of data races. These data races can be corrected by using lock variables in appropriate places in the code.

5. Case Study: Model Checking the Conjugate Gradient Kernel of NPB

This section presents a case study on verifying the data race-freedom and deadlock-freedom of a UPC implementation of the CG kernel of the NPB benchmark (taken from [24]). The NPB benchmarks provide a set of core example applications, called *kernels*, that are used to evaluate the performance of highly parallel supercomputers. The kernels of NPB test different types of applications in terms of their patterns of data access and inter-thread communication. Since the PGAS model is appropriate for solving problems that have irregular data accesses, we select the CG kernel of NPB that implements the conjugate gradient method by the inverse power method (which has an irregular data access pattern). Another interesting feature of CG is the use of a two-dimensional array in the affinity of each thread and the way array cells are accessed. To the best of our knowledge, this section presents the first attempt at mechanical verification of CG. Figure 6 demonstrates the inter-thread synchronization functionalities of CG.

```
1  struct cg_reduce_s { double v[2][NUM_PROC_COLS]; };
2  typedef struct cg_reduce_s cg_reduce_t;
3  shared cg_reduce_t sh_reduce[THREADS];
4
5  int main(int argc, char **argv) {
6     // declaration of local variable
7     // Initialization
8     upc_barrier;
9    // Perform one (untimed) iteration to
10   // initialize code and data page tables
11   /* The call to the conjugate gradient routine */
12    conj_grad( . . . );
13    reduce_sum_2( . . . );
14   // post-processing
15   upc_barrier;
16   /* Main Iteration for inverse power method */
17   for (it = 1; it <= NITER; it++)  {
18        conj_grad( . . . );
19        reduce_sum_2( . . . );
20       // post-processing
21         }
22   upc_barrier;
23   // Thread 0 prints the results
24}
25
26 void reduce_sum_2( double rs_a, double rs_b ) {
27  int rs_i;
28  int rs_o;
29
30  upc_barrier;
31  rs_o = (nr_row * NUM_PROC_COLS);
32  for( rs_i=rs_o; rs_i<(rs_o+NUM_PROC_COLS); rs_i++ ){
33     sh_reduce[rs_i].v[0][MYTHREAD-rs_o] = rs_a;
34     sh_reduce[rs_i].v[1][MYTHREAD-rs_o] = rs_b;    }
35  upc_barrier;
36}
```

Figure 6. Inter-thread synchronization functionalities of the Conjugate Gradient (CG) kernel of the NPB benchmarks.

The first three lines in Figure 6 define a data structure that is used to store the results of computations in a collective reduce fashion. The cg_reduce_s structure captures a two dimensional vector, and the shared array sh_reduce defines a two dimensional vector in the affinity of each thread. After performing some local initializations, all threads synchronize using upc_barrier in Line 8 of Figure 6. Then an untimed iteration of the inverse power method is executed (Lines 9-10) before all threads synchronize again. The reduce_sum_2 routine distributes the results in the shared address space. The for-loop in Line 17 implements the inverse power method. Afterwards, all threads synchronize in Line 22, and then Thread 0 prints out the results. The main difficulty is in the way we abstract the pattern of data accesses in reduce_sum_2.

Abstraction. To capture the way write operations are performed, we consider the following abstract data structures in the Promela model corresponding to sh_reduce:

```
1 typedef bitValStruc { bit b[NUM_PROC_COLS] };
2 typedef bitStruc { bitValStruc v[2] };
3 bitStruc write_sh_reduce[THREADS];
```

Lines 1-2 above define a two dimensional bit array (of type bitStruc) in Promela and Line 3 declares a bit array of bitStruc with size THREADS. Next, we abstract the for-loop of reduce_sum_2 in Promela as follows:

```
1  rso = (_pid / NUM_PROC_COLS) * NUM_PROC_COLS;
2  rsi = rso;
3  do
4    :: (rsi < rso + NUM_PROC_COLS) -> {
5        write_sh_reduce[rsi].v[0].b[_pid-rso] = 1;
6        write_sh_reduce[rsi].v[0].b[_pid-rso] = 0;
7        write_sh_reduce[rsi].v[1].b[_pid-rso] = 1;
8        write_sh_reduce[rsi].v[1].b[_pid-rso] = 0;
9        rsi = rsi + 1; }
10   :: !(rsi < rso + NUM_PROC_COLS) -> break;
11 od;
```

Observe that any write operation has been modeled by setting and resetting the corresponding bit in the array write_sh_reduce.

Model checking CG for data race-freedom. Since the results of local computations are written in shared memory, we only verify freedom from simultaneous writes. Specifically, we verify that before setting a bit in the array write_sh_reduce, that bit is not already set by another thread. We insert an *assert* statement (Lines 3 and 7 below) that verifies data race-freedom. While model checking, SPIN verifies that the assertions hold. In the case of CG for 2, 3 and 4 threads, we found no simultaneous writes.

```
1  do
2   :: (rsi < rso + NUM_PROC_COLS) -> {
3        assert(write_sh_reduce[rsi].v[0].b[_pid-rso] != 1);
4       write_sh_reduce[rsi].v[0].b[_pid-rso] = 1;
5       write_sh_reduce[rsi].v[0].b[_pid-rso] = 0;
6
7        assert(write_sh_reduce[rsi].v[1].b[_pid-rso] != 1);
8       write_sh_reduce[rsi].v[1].b[_pid-rso] = 1;
9       write_sh_reduce[rsi].v[1].b[_pid-rso] = 0;
10      rsi = rsi + 1; }
11  :: !(rsi < rso + NUM_PROC_COLS) -> break;
12 od;
```

Model checking CG for deadlock-freedom. To make sure that each thread eventually terminates, we insert a label "fin:" for a null operation *skip* as the last operation in the body of each thread (similar to the label P in Line 50 of Figure 4). Then we define the following macro:

```
1 #define fin_0  (main[0]@fin)
```

We use similar macros to specify a progress property as $\Diamond$ fin_i for each thread i stating that each thread will eventually terminate; i.e., will not deadlock. SPIN verified that each thread terminates for instances of the CG kernel with 2, 3 and 4 threads.

6. Experimental Results

In order to give a measure of the time/space cost of model checking, this section presents our experimental results for the model checking of the integer permutation program in Figure 2, the Heat Flow program in Figure 5 and the CG program in Figure 6. The platform of model checking is a HP Tablet PC with an Intel Dual Core Processor T5600 (1.83 GHz) and 1GB memory available for model checking.

Model checking of integer permutation. We have model checked the integer permutation program of Figure 2 for data race-freedom and deadlock-freedom, where $3 \leq \mathsf{THREADS} \leq 5$ (see Table 2).

Property	# Threads	Time (Sec.)	# States
Data Race-Freedom	3	0.047	33,189
Data Race-Freedom	4	5.11	2,656,001
Deadlock-Freedom	3	0.125	67,296
Deadlock-Freedom	4	12.4	2,681,440

Table 2. Time/Space costs of the model checking of Integer Permutation.

Verifying data race-freedom took 0.047 and 5.11 seconds respectively for models with 3 and 4 threads. The model checking of thread termination needs more time since the algorithm for the model checking of progress properties is more complex; we spent 0.125 and 12.4 seconds for instances with 3 and 4 threads. SPIN explored 2.7 million states in the model checking of integer permutation for 4 threads. For $\mathsf{THREADS} = 5$, model checking of both data race-freedom and deadlock-freedom took much longer and eventually failed due to insufficient memory.

Model checking of heat flow. Table 3 summarizes our results for the model checking of HF. For 3 threads and region size 3, the model checking took 0.8 seconds for cases where the model satisfied data race-freedom, called the *success cases*, and 0.171 Sec. for cases where the model violates the data race-freedom, called the *failure cases* (e.g., accessing the array cells that are on the boundary of two neighboring regions). Once a failure is found, SPIN stops and provides a counterexample, thereby avoiding the exploration of the unexplored reachable states. For this reason, the verification of failure cases takes less time. For 4 threads and region size 3, model checking took 20.5 seconds for success cases and 0.171 for failure cases. For 5 threads, SPIN failed to give any results due to memory constraints.

To investigate the impact of the region size on the time complexity of model checking, we increased the region size for the case of $\mathsf{THREADS} = 3$ (see Table 3). The time increase in failure cases is negligible. Observe that, increasing the number of threads raises the model checking time exponentially, whereas the region size causes an almost linear increase. This is because increasing the number of threads exponentially increases the number of reachable states due to the combinatorial nature of all possible thread interleavings.

# Threads	Region Size	Time (Sec.)	# States
3	3	0.8	166,740
3	4	2.14	440,852
3	5	5	960,248
3	6	5.19	1,838,224
3	7	9.53	3,209,852
3	8	16.9	5,231,708
3	9	25.1	8,081,872
3	10	37.5	11,959,928
4	3	20.5	6,680,083

Table 3. Time/Space costs of the model checking of Heat Flow for data race-freedom.

In terms of space complexity, SPIN explored almost 11.96 million states for the model checking of an instance of the heat flow model with 3 threads and the region size of 10. Nonetheless, for an instance with 4 threads and region size 3, SPIN explored 6.7 million states. The verification of an instance with 5 threads and the region size 3 failed due to insufficient memory.

Model checking of CG. For 2 threads, the time spent for the model checking of assertion violation is negligible. For deadlock-freedom of each thread, SPIN needed 0.015 seconds. The model checking of assertion violations for 3 and 4 threads respectively required 0.031 and 3.17 seconds and SPIN spent 0.4 and 17.5 for the verification of deadlock-freedom for 3 and 4 threads respectively (see Table 4). The model checking of the CG model with 5 threads was inconclusive due to memory constraints. To verify an instance with 4 threads, SPIN explored almost 3 million states.

Property	# Threads	Time (Sec.)	# States
Data Race-Freedom	3	0.031	19,894
Data Race-Freedom	4	3.17	1,299,858
Deadlock-Freedom	3	0.4	45,282
Deadlock-Freedom	4	17.5	2,988,522

Table 4. Time/Space costs of the model checking of Conjugate Gradient.

7. Conclusions and Future Work

We presented a framework, called UPC-SPIN (see Figure 1), for the model checking of UPC programs. The proposed framework requires programmers to create a tabled abstraction file that specifies how different UPC constructs should be modeled in Promela [10], which is the modeling language of the SPIN model checker [11]. We presented a set of built-in rules that enable the abstraction of UPC synchronization primitives in Promela. The UPC-SPIN framework includes a front-end compiler, called the UPC Model Extractor (UPC-ModEx), that generates finite models of UPC programs, and a back-end that uses SPIN to verify models of UPC programs for properties of interest. Using UPC-SPIN, we have verified several real-world UPC programs including parallel bubble sort, heat flow in metal rods, integer permutation and parallel data collection (see [6] for details). Our verification attempts have both mechanically verified the correctness of programs and have also revealed several concurrency failures (i.e., data races and deadlocks/livelocks). For instance, we have detected data races in a program that models heat flow in metal rods (see Section 4.1). More importantly, we have generated a finite model of a UPC implementation of the Conjugate Gradient (CG) kernel of the NAS Parallel Benchmarks (NPB) [24], and have mechanically demonstrated its correctness for data race-freedom and deadlock-freedom. We have illustrated that even though we verify models of UPC programs with a few threads, it is difficult to *manually* detect the concurrency failures that are detected by the UPC-SPIN framework. Moreover, since SPIN exhaustively checks all reachable states of a model, such failures certainly exists in model instances with larger numbers of threads.

There are several extension to this work. First, we would like to devise abstractions rules for all UPC collectives and implement them in UPC-ModEx, which is the model extractor of UPC-SPIN. Second, to scale up the time/space efficiency of model checking, we are currently working on integrating a swarm platform for model checking [15] in the UPC-SPIN framework so that we can exploit the processing power of computer clusters for the model checking of UPC applications. Third, we plan to investigate the model checking of UPC programs in the relaxed memory consistency model. Last but not least, we believe that a similar approach can be taken to facilitate the model checking of other PGAS languages.

Acknowledgments

The author would like to thank Professor Steve Seidel for his insightful comments about the semantics of UPC constructs and for providing some of the example UPC programs.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] J. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *Proceedings of International Conference on Information Processing – UNESCO*, pages 125–132, 1959.

[3] D. Bozdag, A. H. Gebremedhin, F. Manne, E. G. Boman, and Ü. V. Çatalyürek. A framework for scalable greedy coloring on distributed-memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535, 2008.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[5] U. Consortium. UPC language specifications v1.2, 2005. Lawrence Berkeley National Lab Tech Report LBNL-59208 (`http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf`).

[6] A. Ebnenasir. UPC-SPIN: A Framework for the Model Checking of UPC Programs. Technical Report CS-TR-11-03, Computer Science Department, Michigan Technological University, Houghton, Michigan, July 2011. URL `http://www.cs.mtu.edu/html/tr/11/11-03.pdf`.

[7] T. A. El-Ghazawi and L. Smith. UPC: Unified Parallel C. In *The ACM/IEEE Conference on High Performance Networking and Computing*, page 27, 2006.

[8] E. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B. V., 1990.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : portable parallel programming with the message- passing interface*. MIT Press, 1994.

[10] G. J. Holzmann. Promela language reference. `http://spinroot.com/spin/Man/promela.html`.

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

[12] G. J. Holzmann. Logic verification of ANSI-C code with SPIN. In *7th International Workshop on SPIN Model Checking and Software Verification*, pages 131–147, 2000.

[13] G. J. Holzmann and M. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.

[14] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *International Conference on Software Engineering (ICSE)*, pages 597–607, 1999.

[15] G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2008.

[16] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGARCH Computer Architecture News*, 36(1):329–339, 2008.

[17] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A self-stabilizing approximation algorithm for the maximum matching problem. In *10th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 94–108, 2008.

[18] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0792393805.

[19] 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*, 15(4):391–411, 1997. ISSN 0734-2071.

[20] S. F. Siegel. Verifying parallel programs with MPI-Spin. In *14th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 13–14, 2007.

[21] S. F. Siegel and L. F. Rossi. Analyzing blobflow: A case study using model checking to verify parallel scientific software. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 274–282, 2008.

[22] S. F. Siegel and T. K. Zirkel. Automatic formal verification of MPI-based parallel programs. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 309–310, 2011.

[23] A. Thorsen, P. Merkey, and F. Manne. Maximum weighted matching using the partitioned global address space model. In *HPC '09: Proceedings of the High Performance Computing and Simulation Symposium*, 2009.

[24] UPC NAS Parallel Benchmarks. George Washington University, High-Performance Computing Laboratory. `http://threads.hpcl.gwu.edu/sites/npb-upc`.

[25] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. In *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, pages 3:1–3:6, 2008.