A scalable deadlock detection algorithm for UPC collective operations

Indranil Roy, Glenn R. Luecke, James Coyle, and Marina Kraeva
Iowa State University's High Performance Computing Group, Iowa State University,
Ames, Iowa 50011, USA
Email: iroy@iastate.edu, grl@iastate.edu, jjc@iastate.edu, kraeva@iastate.edu.

***Abstract*—Unified Parallel C (UPC) is a language used to write parallel programs for shared and distributed memory parallel computers. Deadlock detection in UPC programs requires detecting deadlocks that involve either locks, collective operations, or both. In this paper, a distributed deadlock detection algorithm for UPC programs that uses run-time analysis is presented. The algorithm detects deadlocks in collective operations using a distributed technique with O(1) run-time complexity. The correctness and optimality of the algorithm is proven. For completeness, the algorithm is extended to detect deadlocks involving both locks and collective operations by identifying insolvable dependency chains and cycles in a shared wait-for-graph (WFG). The algorithm is implemented in the run-time error detection tool UPC-CHECK and tested with over 150 functionality test cases. The scalability of this deadlock detection algorithm for UPC collective operations is experimentally verified using up to 8192 threads.**

***Keywords*-deadlock, collective, verification, Partitioned Global Address Space (PGAS), Unified Parallel C(UPC).**

I. INTRODUCTION

Unified Parallel C (UPC) [1], [2] is an extension of the C programming language for parallel execution on shared and distributed memory parallel machines. UPC uses the Partitioned Global Address Space (PGAS) [3] parallel programming model where shared variables may be directly read and written by any thread.

Deadlocks in complex application programs are often difficult to locate and fix. Currently UPC-CHECK [4] and UPC-SPIN [5] are the only tools available for the detection of deadlocks in UPC programs. UPC-SPIN employs a model-checking method which inherently does not scale beyond a few threads. In addition, every time the program is modified, the model has to be updated. In contrast, UPC-CHECK uses the algorithm presented in this paper to automatically detect deadlocks at run-time for programs executing on thousands of threads.

This new algorithm not only detects deadlocks involving UPC collective operations, but also verifies the arguments passed to the collective operation for consistency. The run-time complexity of this algorithm is shown to be O(1). The algorithm has been extended to detect deadlocks involving both collective operations and locks. The run-time complexity of the extended algorithm is O(T), where T is the number of threads. Using this deadlock detection algorithm UPC-CHECK detects all deadlock error test cases from the UPC RTED test suite [6].

The rest of this paper is organized as follows. Section II provides the background of various existing deadlock detection techniques. In Section III, a new algorithm to detect potential deadlocks due to incorrect usage of UPC collective operations is presented. The correctness and run-time complexity analysis of the algorithm are also provided. Section IV describes the extended algorithm to detect deadlocks involving both locks and collective operations. The scalability of this deadlock detection algorithm is experimentally confirmed in Section V. Finally, Section VI contains the concluding remarks.

II. BACKGROUND

Out-of-order calls to collective operations on different threads may create a deadlock. Even when the calls to collective operations are in-order, various non-local semantics dictate that consistent arguments need to be used in all participating threads. Non-adherence to these semantics could lead to a deadlock or departure from intended behavior of the program. However, building scalable tools to detect such errors remains a challenge. Träff et al. [7] provided the first verification tool for NEC MPI which used profiling to provide limited non-local checks for parameters like unique *root* thread, operators, length of data etc. Falzone et al. [8] extended these checks to detect errors in datatype signature of parameters using the "datatype signature hashing" mechanism devised by Gropp [9].

Model-checking tools like MPI-SPIN [10] and UPC-SPIN [5] can detect all possible deadlock conditions arising from all combination of parameters in all possible control-flows. However, such tools cannot scale beyond a few threads due to the combinatorial state-space explosion. Tools employing dynamic formal verification methods do not check all the control flows and hence can be used for larger programs. Such tools ISP [11], MODIST [12] and POE [13] generally employ centralized deadlock detection schemes which limit them to verifying executions using a small number of processes. Execution time of such methods is also usually high. DAMPI [14] is a dynamic formal verification tool which overcomes this limitation by using a distributed heuristics-based deadlock detection algorithm.

The most practical method for detecting deadlocks in terms of scalability is run-time analysis. Tools using this kind of analysis only detect deadlocks which would actually occur during the current execution of a program. Marmot [15] and MPI-CHECK [16] employ synchronized time-out based strategies to detect deadlock conditions. Time-out based strategies may report false-positive error cases and generally cannot pinpoint the exact reason for the error. On the other hand, the run-time analysis tool, Umpire [17] uses a centralized WFG based on the generalized $AND \oplus OR$ model developed by Hilbrich et al. [18]. However, MPI-CHECK, Marmot and Umpire are all based on the client-server model, which limits their scalability to a few hundred threads. In order to overcome this limitation, MUST [19] utilizes a flexible and efficient communication system to transfer records related to error detection between different processes or threads.

Our algorithm uses a different approach to detect deadlocks involving collective operations. We exploit two properties of operations in UPC which make deadlock detection easier than in MPI. Firstly, communication between two processes is non-blocking and secondly, non-determinism of point-to-point communication operations in terms of any_source cannot occur in UPC. However, both UPC and MPI require that the order of collective operations and the values passed to the single-valued arguments must be the same on all threads/processes. Non-adherence to these restrictions could lead to a deadlock. We extend our algorithm to detect deadlocks involving locks, collective operations and both by using a distributed shared WFG. In our WFG, we identify not only dependency cycles but also those *dependency chains* that cannot be satisfied due to blocking collective operations.

III. DETECTING DEADLOCKS DUE TO COLLECTIVE ERRORS IN COLLECTIVE OPERATIONS

Terms used throughout the rest of this paper are:

1) $THREADS$ is an integer variable that refers to the total number of threads with which the execution of the application was initiated.
2) A UPC *operation* is defined as any UPC statement or function listed in the UPC specification.
3) The *state* of a thread is defined as the name of the UPC operation that the thread has reached. In case the thread is executing an operation which is not a collective or lock-related UPC operation, the *state* is set to `unknown`. If the thread has completed execution, the *state* is set to `end_of_execution`.
4) A *single-valued* argument is an argument of a UPC collective operation which must be passed the same value on every thread.
5) The *signature* of a UPC operation on a thread consists of the name of the UPC operation and the values which are about to be passed to each of the single-valued arguments of the UPC collective operation on that thread.
6) For any thread k, s_k is a shared data structure which stores the state of thread k in field $s_k.op$. In case state is the name of a UPC collective operation, s_k also stores the single-valued arguments of the operation on that thread.
7) To *compare* the signatures of UPC operations stored in s_i and s_j means to check whether all the fields in s_i and s_j are identical.
8) If all the fields in s_i and s_j are identical, the result of the comparison is a *match*, otherwise there is a *mismatch*.
9) $C(n, k)$ denotes the n^{th} collective operation executed by thread k.

The UPC specification requires that the order of calls to UPC collective operations must be the same for all threads [20]. Additionally, each '*single-valued*' argument of a collective operation must have the same value on all threads. Therefore deadlocks involving only collective UPC operations can be created if:

1) different threads are waiting at different collective operations,
2) values passed to single-valued arguments of collective functions do not match across all threads, and
3) some threads are waiting at a collective operation while at least one thread has finished execution.

An algorithm to check whether any of the above 3 cases is going to occur must compare the collective operation which each thread is going to execute next and its single-valued arguments with those on other threads. Our algorithm achieves this by viewing the threads as if they were arranged in a circular ring. The left and right neighbors of a thread i are thread $(i-1)\%THREADS$ and thread $(i+1)\%THREADS$ respectively. Each thread checks whether its right neighbor has reached the same collective operation as itself. Since this checking goes around the whole ring, if all the threads arrive at the same collective operation, then each thread will be verified by its left neighbor and there will be no mismatches of the collective operations. However, if any thread comes to a collective operation which is not the same as that on the other thread, its left neighbor can identify the discrepancy, and issue an error message. This is illustrated in Figure 1. The correctness of this approach is proven in Section III-C.

On reaching a collective UPC operation, a thread k first records the signature of the collective operation in s_k. Thread k sets $s_k.op$ to `unknown` after exiting from a operation. Let a and b be the variables that store signatures of collective operations. The assign ($\leftarrow$) and the compare ($\ncong$) operations for the signatures of collective operation stored in a and b are defined as follows:

1) $b \leftarrow a$ means
    a) assign value of variable $a.op$ to variable $b.op$, and
    b) if $a.op \neq end\_of\_execution$, copy values of

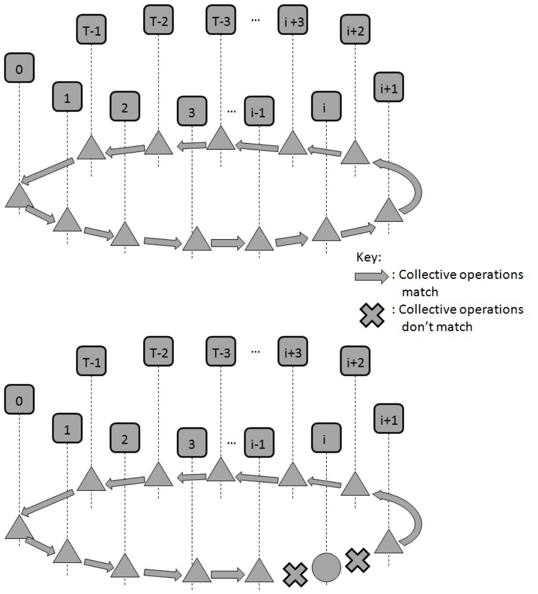

Figure 1. Circular ring of threads checking the order of collective UPC operations

single-valued arguments recorded in a to b

2) $b \ncong a$ is true if
   a) $b.op \neq a.op$, or
   b) if $a.op \neq end\_of\_execution$, any of the single-valued arguments recorded in a is not identical to the corresponding argument recorded in b.

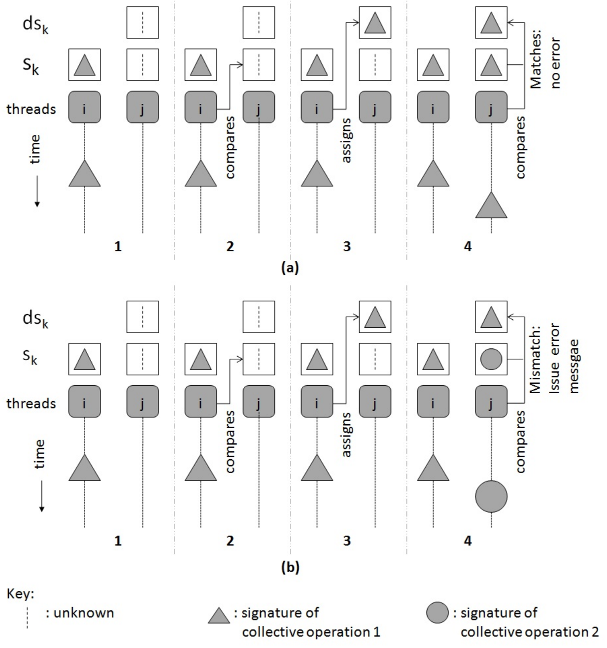

Figure 2. Checking *signatures*: Thread i reaches collective operation before thread j. (a) no error case. (b) error case.

Let thread j be the right neighbor of thread i. During execution, thread i or thread j could reach their respective n^{th} collective operation first. If thread i reaches the operation first, then it cannot compare $C(n,i)$ recorded in s_i with $C(n,j)$, since s_j does not contain the signature of the n^{th} collective operation encountered on thread j, i.e. $C(n,j)$. The comparison can be delayed until thread j reaches its n^{th} collective operation. In order to implement this, another shared variable ds_k is used on each thread k to store the desired signature. For faster access, both shared variables s_k and ds_k have *affinity*[1] to thread k. If thread i finds that thread j has not reached a collective operation ($s_j.op$ is `unknown`), then it assigns s_i to ds_j. When thread j reaches a collective operation it first records the signature in s_j and then compares it with ds_j. If they do not match, then thread j issues an error message, otherwise it sets $ds_j.op$ to `unknown` and continues. This is illustrated in Figure 2.

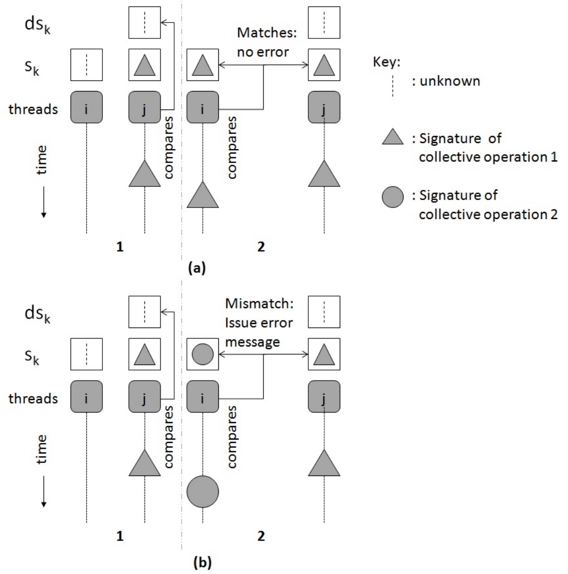

Figure 3. Checking *signatures*: Thread i reaches collective operation after thread j. (a) no error case. (b) error case.

If thread i reaches the collective operation after thread j ($s_j.op$ is assigned a name of a collective UPC operation), then thread i compares s_j with s_i. If they match, then there is no error, so execution continues. This is illustrated in Figure 3.

The UPC specification does not require collective operations to be synchronizing. This could result in one or more state variables on a thread being reassigned with the signature of the next collective operation that it encounters before the necessary checking is completed. To ensure that the signature of the n^{th} collective operation encountered on thread i i.e. $C(n,i)$ is compared with the signature of the n^{th}

---

[1] In UPC, shared variables that are stored in the physical memory of a thread are said to have *affinity* to that thread.

collective operation encountered on thread j, i.e. $C(n, j)$, the algorithm must ensure that:

1) If thread i reaches the n^{th} collective operation before thread j and assigns ds_j the signature of $C(n, i)$, it does not reassign ds_j before thread j has compared ds_j with s_j, and
2) If thread j reaches the n^{th} collective operation before thread i and assigns s_j the signature of $C(n, j)$, it does not reassign s_j before either thread i has a chance to compare it with s_i or thread j has a chance to compare it with ds_j.

In order to achieve the behavior described above, two shared variables r_s_j and r_ds_j are used for every thread j. Variable r_s_j is used to prevent thread j from reassigning s_j before the necessary comparisons described above are completed. Similarly, variable r_ds_j is used to prevent thread i from reassigning ds_j before the necessary comparisons are completed. Both r_s_j and r_ds_j have affinity to thread j.

For thread j, shared data structures s_j and ds_j are accessed by thread i and thread j. To avoid race conditions, accesses to s_j and ds_j are guarded using lock $L[j]$.

Our deadlock algorithm is implemented via the following three functions:

- *check_entry()* function which is called before each UPC operation to check whether executing the operation would cause a deadlock,
- *record_exit()* function which is called after each UPC operation to record that the operation is complete and record any additional information if required, and
- *check_final()* function which is called before every `return` statement in the `main()` function and every `exit()` function to check for possible deadlock conditions due to the termination of this thread.

The pseudo-code of the distributed algorithm[2] on each thread i to check deadlocks caused by incorrect or missing calls to collective operations[3] is presented below. Function *check_entry*() receives as argument the signature of the collective operation that the thread has reached, namely f_sig.

A. Algorithm A1: Detecting wrong-order sequence of calls to collective operations

```
On thread $i$:
----------------------------------------------------------
Initialization
$s_i.op \leftarrow ds_i.op \leftarrow unknown$, $r\_s_i \leftarrow 1$, $r\_ds_j \leftarrow 1$
----------------------------------------------------------
{Function definition of check_entry(f_sig):}
if $THREADS$ = 1 then
    Exit check.
else
    Acquire $L[i]$
    $s_i \leftarrow f\_sig$
    $r\_s_i \leftarrow 0$
    if $ds_i.op \neq unknown$ then

        if $ds_i \ncong s_i$ then
            Print error and call global exit function.
        end if
        $r\_s_i \leftarrow 1$
        $r\_ds_i \leftarrow 1$
        $ds_i.op \leftarrow unknown$
    end if
    Release $L[i]$
    Wait until $r\_ds_j = 1$
    Acquire $L[j]$

    if $s_j.op = unknown$ then

        $ds_j \leftarrow s_i$
        $r\_ds_j \leftarrow 0$
    else
        if $s_j \ncong s_i$ then
            Print error and call global exit function
        end if
        $r\_s_j \leftarrow 1$
    end if
    Release $L[j]$
end if
----------------------------------------------------------
{Function definition of check_exit():}
Wait until $r\_s_i = 1$
Acquire $L[i]$
$s_i.op \leftarrow unknown$
Release $L[i]$
----------------------------------------------------------
{Function definition of check_final():}
Acquire $L[i]$
if $ds_i.op \neq unknown$ then
    Print error and call global exit function.
end if
$s_i.op \leftarrow end\_of\_execution$
Release $L[i]$
----------------------------------------------------------
```

B. Detecting deadlock errors involving upc_notify and upc_wait operations

The compound statement {*upc_notify; upc_wait*} forms a split barrier in UPC. The UPC specification requires that firstly, there should be a strictly alternating sequence of

---

[2] As presented, the algorithm forces synchronization even for non-synchronizing UPC collective operations. However, if forced synchronization is a concern, this can be handled with a queue of states. This will not change the O(1) behavior.

[3] UPC-CHECK treats non-synchronizing collective operations as synchronizing operations because the UPC 1.2 specification says that "Some implementations may include unspecified synchronization between threads within collective operations" (footnote; page 9).

upc_notify and upc_wait calls, starting with a upc_notify call and ending with a upc_wait call. Secondly, there can be no collective operation between a upc_notify and its corresponding upc_wait call. These conditions are checked using a private binary flag on each thread which is set when a upc_notify statement is encountered and reset when a upc_wait statement is encountered. This binary flag is initially reset. If any collective operation other than upc_wait is encountered when the flag is set, then there must be an error. Similarly, if a upc_wait statement is encountered when the flag is reset, then there must be an error. Finally, if the execution ends, while the flag is set, then there must be an error. These checks are performed along with the above algorithm and do not require any communication between threads. Also modifying and checking private flags is an operation with complexity of O(1).

If all the threads issue the upc_notify statement, then the next UPC collective operation issued on all the threads must be a upc_wait statement. Therefore algorithm $A1$ working in unison with the above check needs to only verify the correct ordering of upc_notify across all threads. The correct ordering of the upc_wait statements across all threads is automatically guaranteed with the above mentioned checks. This is reflected in Algorithm $A2$.

C. Proof of Correctness

Using the same relation between thread i and thread j, i.e. thread i is the left neighbor of thread j, the proof of correctness is structured as follows. Firstly, it is proved that the algorithm is free of deadlocks and livelocks. Then Lemma 3.1 is used to prove that the left neighbor of any thread j does not reassign ds_j before thread j can compare s_j with ds_j. Lemma 3.2 proves that the right neighbor of any thread i, does not reassign s_j before thread i can compare s_i with s_j. Using Lemma 3.1 and Lemma 3.2 it is proven that for any two neighboring threads i and j, signature of $C(n, j)$ is compared to the signature of $C(n, i)$. Finally, using Lemma 3.3 the correctness of the algorithm is proven by showing that : 1) no error message is issued if all the threads have reached the same collective operation with the same signature and 2) an error message is issued if at least one thread has reached a collective operation with a signature different from the signature of the collective operation on any other thread. Case 1 is proved by Theorem 3.4 and Case 2 is proved by Theorem 3.5.

There is no hold-and-wait condition in algorithm A1, hence there cannot be any deadlocks in the algorithm. To show that the algorithm is livelock-free, we show that any given thread must eventually exit the waits on line 24 and 42. For any thread i reaching its n^{th} collective operation $C(n, i)$, thread i can wait at line 24 if thread i itself had set r_ds_j to 0 on line 30 on reaching $C(n-1, i)$. This is possible only if thread i found that $s_j.op = unknown$ on line 27, i.e. thread j is not executing an UPC collective operation. Eventually thread j either reaches the end of execution or a UPC collective operation. In the former case, a deadlock condition is detected, an error message is issued and the application exits. In the second case, thread j finds conditional statement on line 14 to be true and sets r_ds_j to 1 on line 20. Since only thread i can set r_ds_j to 0 again, thread i would definitely exit the wait on line 24. Similarly, for thread j to be waiting at line 42 after executing $C(n, j)$, it must not have set r_s_j to 1 at line 19. This means that $ds_j.op$ must be equal to $unknown$ at line 14, implying that thread i has still not executed line 29 and hence line 27 (by temporal ordering) due to the atomic nature of operations accorded by $L[j]$. When thread i finally acquires $L[j]$, the conditional statement on line 27 must evaluate to false. If thread i has reached a collective operation with a signature different from that of $C(n, j)$, a deadlock error message is issued, otherwise r_s_j is set to 1. Since only thread j can set r_s_j to 0 again, it must exit the waiting at line 42.

Lemma 3.1: After thread i assigns the signature of $C(n, i)$ to ds_j, then thread i does not reassign ds_j before thread j compares s_j with ds_j.

Proof: This situation arises only if thread i has reached a collective operation first. After thread i sets ds_j to s_i (which is already set to $C(n, i)$) at line 29, it sets r_ds_j to 0 at line 30. Thread i cannot reassign ds_j until r_ds_j is set to 1. Only thread j can set r_ds_j to 1 after comparing s_j with ds_j at line 21. ■

Lemma 3.2: After thread j assigns the signature of $C(n, j)$ to s_j, then thread j does not reassign s_j before it is compared with s_i.

Proof: After thread j assigns the signature of $C(n, j)$ to s_j at line 13, it sets r_s_j to 0. Thread j cannot modify s_j until r_s_j is set to 1. If thread i has already reached the collective operation, then thread j sets r_s_j to 1 at line 20 only after comparing s_j with ds_j at line 17. However, thread i must have copied the value of s_i to ds_j at line 29. Alternatively, thread j might have reached the collective operation first. In this case, thread i sets r_s_j to 1 at line 36 after comparing s_i to s_j at line 33. ■

Lemma 3.3: For any neighboring threads i and j, the signature of $C(n, i)$ is always compared with the signature of $C(n, j)$.

Proof: This is proved using induction on the number of the collective operations encountered on threads i and j.

Basis. Consider the case where n equals 1, i.e. the first collective operation encountered on thread i and thread j. The signature of $C(1, i)$ is compared with the signature of $C(1, j)$. If thread i reaches collective operation $C(1, i)$ first, then it assigns ds_j the signature of $C(1, i)$. Using Lemma 3.1, thread i cannot reassign ds_j until ds_j is compared with s_j by thread j on reaching its first collective operation, $C(1, j)$. Alternatively, if thread j reaches its collective operation first, then Lemma 3.2 states that after thread j assigns the signature of $C(1, j)$ to s_j, thread

j cannot reassign s_j before it is compared with s_i. The comparison between s_j and s_i is done by thread i after it reaches its first collective operation and has assigned s_i the signature of $C(1, i)$.

Inductive step. If the signature of $C(n, i)$ is compared with the signature of $C(n, j)$, then it can be proven that the signature of $C(n+1, i)$ is compared with the signature of $C(n+1, j)$. If thread i reaches its next collective operation $C(n+1, i)$ first, then it assigns ds_j the signature of $C(n+1, i)$. Using Lemma 3.1, thread i cannot reassign ds_j until ds_j is compared with s_j by thread j on reaching its next collective operation, i.e. $C(n+1, j)$. Alternatively, if thread j reaches its next collective operation first, then Lemma 3.2 states that after thread j assigns $C(n+1, j)$ to s_j, thread j cannot reassign s_j before it is compared with s_i. The comparison of s_j with s_i is done by thread i after it reaches its next collective operation and has asigned s_i the signature of $C(n+1, i)$. ■

Using Lemma 3.3, it is proven that for any neighboring thread pair i and j, the signature of n^{th} collective operation of thread i is compared with the signature of n^{th} collective operation of thread j. As j varies from 0 to $THREADS-1$, it can be said that when the n^{th} collective operation is encountered on any thread, it is checked against the n^{th} encountered collective operation on every other thread before proceeding. Thus in the following proofs, we need to only concentrate on a single (potentially different) collective operation on each thread. In the following proofs, let the signature of the collective operation encountered on a thread k be denoted by S_k. If a state or desired state $a_i.op$ is `unknown`, then it is denoted as $a = U$ for succinctness. Then in algorithm $A1$, after assigning the signature of the encountered collective operation, i.e. line $s_i \leftarrow f\_sig$, notice that for thread i:
s_i must be S_i,
ds_i must be either U or S_{i-1},
s_j must be either U or S_j, and
ds_j must be U.

Theorem 3.4: If all the threads arrive at the same collective operation, and the collective operation has the same signature on all threads, then Algorithm $A1$ will not issue an error message.

Proof: If $THREADS$ is 1, no error message is issued, so we need to consider only cases of execution when $THREADS > 1$. If all threads arrive at the same collective operation with the same signature, then during the checks after $s_i \leftarrow f\_sig$, is the same for all i. Let S denote this common signature. We will prove this theorem by contradiction. An error message is printed only if:

1) $ds_i \neq U$ and $ds_i \neq s_i \Rightarrow ds_i = S$ and $ds_i \neq S \Rightarrow S \neq S$ (contradiction) or
2) $s_j \neq U$ and $s_j \neq s_i \Rightarrow s_j = S$ and $s_j \neq S \Rightarrow S \neq S$ (contradiction)

So Theorem 3.4 is proved. ■

Theorem 3.5: If any thread has reached a collective operation with a signature different from the signature of the collective operation on any other thread, then a deadlock error message is issued.

Proof: There can be a mismatch in the collective operation or its signature only if there is more than one thread.

Since the signatures of the collective operations reached on every thread are not identical, there must be some thread i for which $S_i \not\cong S_j$. For these threads i and j, the following procedures are made to be atomic and mutually exclusive through use of lock $L[j]$:

- Action 1: Thread i checks s_j. If $s_j = U$, then thread i executes $ds_j \leftarrow s_i$, else, computes $s_j \not\cong s_i$ and issues an error message if true.
- Action 2: Thread j assigns the signature of the collective operation it has reached to s_j. Thread j checks ds_j. If $ds_j \neq U$, the thread j computes $ds_j \not\cong s_j$ and issues message if true.

There are only two possible cases of execution: either action 1 is followed by action 2 or vice versa.

In the first case, in action 1, thread i finds $s_j = U$ is true, executes $ds_j \leftarrow S_i$ and continues. Then in action 2, thread j executes $s_j \leftarrow S_j$, finds that $ds_j \neq U$ and hence computes $ds_j \not\cong s_j$. Now, since $ds_j = S_i$ and $s_j = S_j$ and $S_i \neq S_j$ (by assumption) implies that $ds_j \not\cong s_j$ is true. Therefore thread j issues an error message.

In the second case, in action 2, thread j assigns $s_j \leftarrow S_j$, finds $ds_j = U$ and continues. Before thread i initiates action 1 by acquiring $L[j]$, it must have executed $s_i \leftarrow S_i$. If $ds_i \neq U$ and $ds_i \not\cong s_i$, then an error message is issued by thread i, otherwise it initiates action 1. Thread i finds $s_j \neq U$ and computes $s_j \not\cong s_i$. Now, since $s_i = S_i$ and $s_j = S_j$ and $S_i \not\cong S_j$ (by assumption) implies that $s_j \not\cong s_j$ is true. Therefore thread i issues an error message.

Since the above two cases are exhaustive, an error is always issued if $S_i \not\cong S_j$ and hence Theorem 3.5 is proved. ■

Theorem 3.6: The complexity of the Algorithm $A1$ is O(1).

Proof: There are two parts to this proof.

1) The execution-time overhead for any thread i is O(1). Any thread i computes a fixed number of instructions before entering and after exiting a collective operation. It waits for at most two locks $L[i]$ and $L[j]$ each of which can have a dependency chain containing only one thread, namely thread $i-1$ and thread j respectively. Thread i synchronizes with only two threads, i.e. its left neighbor thread $i-1$ and right neighbor thread j. There is no access to variables or locks from any other thread. Therefore the execution time complexity of the algorithm in terms of the number of threads is O(1).

2) The memory overhead of any thread i is independent of the number of threads and is constant.

■

IV. DETECTING DEADLOCKS CREATED BY HOLD-AND-WAIT DEPENDENCY CHAINS FOR ACQUIRING LOCKS

In UPC, acquiring a lock with a call to the $upc\_lock()$ function is a blocking operation. In UPC program, deadlocks involving locks occur when there exists one of the following conditions:

1) a cycle of hold-and-wait dependencies with at least two threads, or
2) a chain of hold-and-wait dependencies ending in a lock held by a thread which has completed execution, or
3) a chain of hold-and-wait dependencies ending in a lock held by a thread which is blocked at a synchronizing collective UPC operation.

Deadlocks caused by the hold-and-wait dependencies can be detected using a WFG shown in Figure 4. Threads waiting for a lock are shown using boxes whereas locks are shown as circles. A dashed arrow from a thread to the lock depicts that thread is *waiting* for that lock. A solid arrow from a lock to a thread shows that thread is *holding* that lock.

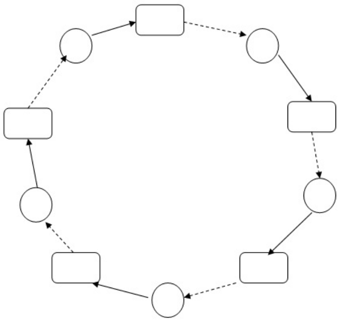

Figure 4. Circular dependencies of threads leading to a deadlock.

Using the same notations for locks, threads, hold and wait actions, Figure 5 illustrates a chain of hold-and-wait dependencies. This chain of dependencies will never be resolved if the lock held by the thread depicted as the gray box will never be released. This can happen only if the thread has either completed execution or is blocked at a synchronizing collective operation which will not be completed.

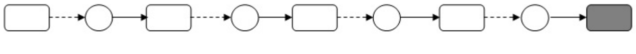

Figure 5. Chain of hold-and wait dependencies while trying to acquire a lock leading to a deadlock.

Our algorithm uses a simple edge-chasing method to detect deadlocks involving locks in UPC programs. Before a thread u tries to acquire a lock, it checks if the lock is free or not. If it is free, the thread continues execution. Otherwise, if the lock is held by thread v, thread u checks $s_v.op$ to check if thread v:

1) is not executing a collective UPC operation or `upc_lock` operation ($s_v.op$ is $unknown$), or
2) is waiting to acquire a lock, or
3) has completed execution, or
4) is waiting at a synchronizing collective UPC operation.

If thread v is waiting to acquire a lock, then thread u continues to check the $state$ of the next thread in the chain of dependencies. If thread u finally reaches thread m which is not executing a collective UPC operation or `upc_lock` operation, then no deadlock is detected. If thread u finds itself along the chain dependencies, then it reports a deadlock condition. Similarly, if thread u finds thread w which has completed execution at the end of the chain of dependencies, then it issues an error message.

When the chain of dependencies ends with a thread waiting at a collective synchronizing operation, the deadlock detection algorithm needs to identify whether the thread will finish executing the collective operation or not. Figure 6 illustrates these two cases. Thread u is trying to acquire a lock in a chain of dependencies ending with thread w. When thread u checks the $s_w.op$ of thread w, thread w may (a) not have returned from the n^{th} synchronizing collective operation $C_s(n, w)$, (b) have returned from the n^{th} synchronizing collective operation but has not updated the $s_w.op$ in the $check\_exit()$ function, (c) have completed executing $check\_entry()$ function for the next synchronizing collective operation $C_s(n+1, w)$, or (d) waiting at the $(n+1)^{th}$ synchronizing collective operation $C_s(n+1, w)$. The n^{th} synchronizing collective operation encountered on thread w must be a valid synchronization operation that all threads must have called (otherwise the $check\_entry()$ function would have issued an error message). Therefore scenarios (a) and (b) are not deadlock conditions, while (c) and (d) are. To identify and differentiate between these scenarios, a binary shared variable $sync\_phase_k$ is introduced for each thread k. Initially $sync\_phase_k$ is set to 0 for all threads. At the beginning of each $check\_entry()$ function on thread k, the value $sync\_phase_k$ is toggled. Thread u can now identify the scenarios by just comparing $sync\_phase_u$ and $sync\_phase_w$. If they match (are *in-phase*), then it is either scenario (a) or (b) and hence no deadlock error message is issued. If they do not match (are *out-of-phase*), then it is either scenario (c) or (d) and hence a deadlock error message is issued.

A. The complete deadlock detection algorithm

The complete algorithm to detect deadlocks created by errors in collective operations and hold-and-wait dependency chains for acquiring locks is presented below. The

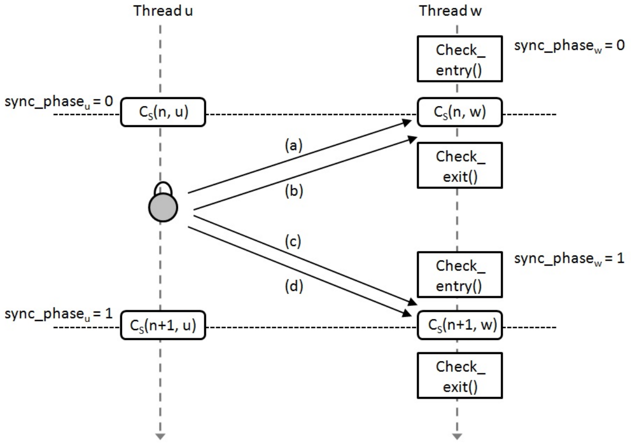

Figure 6. Possible scenarios when detecting deadlocks involving chain of hold-and wait dependencies. Scenario (a) or (b) is not a deadlock condition, while scenario (c) or (d) is.

check_entry() and *check_exit*() functions receive two arguments: 1) the signature of the UPC operation that the thread has reached, namely f_sig and 2) the pointer L_ptr. L_ptr points to the lock which the thread is trying to acquire or release if the thread has reached a *upc_lock*, *upc_lock_attempt* or *upc_unlock* statement.

Algorithm A2.

```
On thread $i$:
------------------------------------------------------------
Initialization
Create empty list of acquired and requested locks
$s_i.op \leftarrow ds_i.op \leftarrow unknown$, $r\_s_i \leftarrow 1$, $r\_ds_j \leftarrow 1$,
($sync\_phase_i \leftarrow 0$)
------------------------------------------------------------
{Function definition of check_entry(f_sig, L_ptr):}
Acquire $L[i]$
$s_i \leftarrow f\_sig$
Release $L[i]$
if $f\_sig.op$ = $at\_upc\_wait\_statement$ then
    Exit check
else if $f\_sig.op$ = $at\_upc\_lock\_operation$ then
    Acquire $c\_L$
    Check status of $L_ptr$
    if $L\_ptr$ is held by this thread
            or is part of a cycle
            or chain of dependencies then
        Print suitable error and call global exit
    else
        Update list of requested locks
        Release $c\_L$
        Exit check
    end if
else if $f\_sig.op$ = $at\_upc\_unlock\_operation$ then
    if $L\_ptr$ is not held by this thread then
        Print suitable error and call global exit.
    else
        Update list of acquired locks
        Exit check
    end if
else
    {Thread must have reached a collective operation}
    if $THREADS$ = $1$ then
        Exit check.
    end if
    Acquire $c\_L$
    if this thread holds locks which are in the list of
    requested locks then
        Print suitable error and call global exit.
    end if
    Release $c\_L$
    Acquire $L[i]$
    $r\_s_i \leftarrow 0$
    if this is a synchronizing collective operation then
        $sync\_phase_i \leftarrow (sync\_phase_i + 1)\%2$
    end if
    if $ds_i.op \neq unknown$ then
        if $ds_i \ncong s_i$ then
            Print error and call global exit function.
        end if
        $r\_s_i \leftarrow 1$
        $r\_ds_i \leftarrow 1$
        $ds_i.op \leftarrow unknown$
    end if
    Release lock $L[i]$
    Wait until $r\_ds_j = 1$
    Acquire lock $L[j]$
    if $s_j.op = unknown$ then
        $ds_j \leftarrow s_i$
        $r\_ds_j \leftarrow 0$
    else
        if $s_j \ncong s_i$ then
            Print error and call global exit function
        end if
        $r\_s_j \leftarrow 1$
    end if
    Release lock $L[j]$
end if
------------------------------------------------------------
{Function definition of $check\_exit(f\_sig, L\_ptr)$:}
Wait until $r\_s_i = 1$
Acquire $L[i]$
$s_i \leftarrow unknown$
Release $L[i]$
if $f\_sig.op = at\_upc\_lock\_operation$ then
    Acquire $c\_L$
    Remove $L\_ptr$ from the list of requested locks
    Add $L\_ptr$ to the list of acquired locks
    Release $c\_L$
```

```
    Continue execution.
else if $f\_sig.op$ = $at\_upc\_lock\_attempt\_operation$ then
    if $L\_ptr$ was achieved then
        Acquire $c\_L$
        Remove $L\_ptr$ from the list of requested locks
        Add $L\_ptr$ to the list of acquired locks
        Release $c\_L$
    end if
    Continue execution.
else
    Continue execution.
end if
----------------------------------------
{Function definition of $check\_final()$:}
Acquire $L[i]$
$s_i \leftarrow end\_of\_execution$
if $ds_i.op \neq unknown$ then
    Print error and call global exit function.
end if
Release $L[i]$
Acquire $c\_L$
if this thread holds locks which are in the list of requested locks then
    Print suitable error and call global exit.
end if
if this thread is still holding locks then
    Print suitable warning
end if
Release $c\_L$
----------------------------------------
```

Checking for dependency chains and cycles adds only a constant amount of time overhead for each thread in the chain or cycle. This means that the overhead is O(T) where T is the number of threads in the dependency chain.

V. EXPERIMENTAL VERIFICATION OF SCALABILITY

This deadlock detection algorithm has been implemented in the UPC-CHECK tool [4]. UPC-CHECK was used to experimentally verify the scalability of this algorithm on a Cray XE6 machine running the CLE 4.1 operating system. Each node has two 16-core Interlagos processors. Since we are interested in the verification of scalability, the authors measured the overhead of our deadlock detection method for 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 and 8192 threads. The verification of scalability was carried out by first measuring the overhead incurred when calling a UPC collective operation and then measuring the overhead when running the CG and IS UPC NAS Parallel Benchmarks (NPB) [21]. The Cray C 8.0.4 compiler was used with the `-hupc` option. To pin processes and memory the aprun command was used with the following options: `-ss -cc cpu`.

The authors first measured the overhead of checking for deadlocks involving the `upc_all_broadcast` operation with a message consisting of one 4 byte integer. Since deadlock checking is independent of the message size, the small message size was used so that the checking overhead could be easily measured. To measure the time accurately, 10,000 calls to `upc_all_broadcast` were timed and an average reported.

```
time (t1);
for (i = 0; i < 10000; i++)
{
    upc_all_broadcast;
}
time {t2};
bcast_time = (t2 - t1)/10000;
```

Overhead times ranged from 76 to 123 microseconds for multiple nodes, i.e. 64, 128, 256, 512, 1024, 2048, 4096 and 8192 threads. When replacing `upc_all_broadcast` with `upc_all_gather_all`, overhead times ranged from 73 to 119 microseconds. In both cases, a slight increase is observed as we increase the number of threads. The authors attribute this to the fact that, in general, not all pairs of UPC threads can be mapped to physical processors for which the communication between UPC threads i and $(i+1)\%THREADS$ is the same for all i. The maximal communication time for optimally placed UPC threads still grows slowly as the total number of UPC threads grows. The deviation from constant time in the above experiment is only a factor of 1.5 for 128 times as many UPC threads.

UPC-CHECK was tested for correctness using 150 tests from the UPC RTED test suite [6]. Each test contains a single deadlock. For all the tests, UPC-CHECK detects the error, prevents the deadlock from happening and exits after reporting the error correctly [4]. Since these tests are very small, the observed overhead was so small that we could not measure them accurately.

Timing results for the UPC NPB CG and IS benchmarks are presented in Tables I and II using 2, 4, 8, 16, 32, 64, 128, and 256 threads. Timings using more than 256 threads could not be obtained since these benchmarks are written in a way that prevents them from being run with more than 256 threads. These results also demonstrate the scalability of the deadlock detection algorithm presented in this paper. Timing data for the class B CG benchmark using 256 threads could not be obtained since the problem size is too small to be run with 256 threads.

VI. CONCLUSION

In this paper, a new distributed and scalable deadlock detection algorithm for UPC collective operations is presented. The algorithm has been proven to be correct and to have a run-time complexity of O(1). This algorithm has been extended to detect deadlocks involving locks with a run-time

Number of threads	**Class B**			**Class C**		
	Without checks	**With checks**	**Overhead**	**Without checks**	**With checks**	**Overhead**
2	77.2	77.6	0.4	211.2	211.8	0.6
4	41.4	41.7	0.3	112.7	112.8	0.1
8	28.1	28.7	0.6	73.9	74.2	0.3
16	15.3	16.0	0.6	39.4	40.0	0.6
32	8.6	9.5	0.9	21.1	22.1	0.9
64	5.5	6.6	1.1	13.1	14.0	1.0
128	3.3	4.7	1.3	8.3	9.7	1.4
256	NA	NA	NA	5.6	7.2	1.6

Table I
TIME IN SECONDS OF THE UPC NPB-CG BENCHMARK WITH AND WITHOUT DEADLOCK CHECKING

Number of threads	**Class B**			**Class C**		
	Without checks	**With checks**	**Overhead**	**Without checks**	**With checks**	**Overhead**
2	4.56	4.59	0.03	20.00	20.11	0.11
4	2.18	2.18	0.00	9.50	9.52	0.01
8	1.34	1.34	0.00	5.28	5.28	0.00
16	0.79	0.79	0.00	3.46	3.46	0.00
32	0.42	0.43	0.01	1.89	1.89	0.00
64	0.29	0.30	0.01	1.30	1.31	0.01
128	0.21	0.22	0.01	0.82	0.82	0.00
256	0.26	0.27	0.01	0.57	0.57	0.00

Table II
TIME IN SECONDS OF THE UPC NPB-IS BENCHMARK WITH AND WITHOUT DEADLOCK CHECKING

complexity of O(T), T is the number of threads involved in the deadlock. The extended algorithm utilizes a distributed technique to check deadlock errors in collective operations and uses a distributed wait-for-graph for detecting deadlocks involving locks. The algorithm has been implemented in the run-time error detection tool UPC-CHECK and tested with over 150 functionality test cases. The scalability of this deadlock detection algorithm has been experimentally verified using up to 8192 threads.

In UPC-CHECK, the algorithm is implemented through automatic instrumentation of the application via a source-to-source translator created using the ROSE toolkit [22]. Alternatively, such error detection capability may be added during the precompilation step of a UPC compiler. This capability could be enabled using a compiler option and may be used during the entire debugging process as the observed memory and execution time overhead even for a large number of threads is quite low.

ACKNOWLEDGMENT

This work was supported by the United States Department of Defense & used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory.

REFERENCES

[1] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*. Wiley-Interscience, 2003.

[2] "Unified Parallel C (Wikipedia)." [Online]. Available: http://en.wikipedia.org/wiki/Unified_Parallel_C

[3] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, ser. PASCO '07. New York, NY, USA: ACM, 2007, pp. 24–32. [Online]. Available: http://doi.acm.org/10.1145/1278177.1278183

[4] J. Coyle, I. Roy, M. Kraeva, and G. Luecke, "UPC-CHECK: a scalable tool for detecting run-time errors in Unified Parallel C," *Computer Science - Research and Development*, pp. 1–7, 10.1007/s00450-012-0214-4. [Online]. Available: http://dx.doi.org/10.1007/s00450-012-0214-4

[5] A. Ebnenasir, "UPC-SPIN: A Framework for the Model Checking of UPC Programs," in *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '11, 2011. [Online]. Available: http://pgas11.rice.edu/papers/Ebnenasir-UPC-Model-Checking-PGAS11.pdf

[6] J. Coyle, J. Hoekstra, M. Kraeva, G. R. Luecke, E. Kleiman, V. Srinivas, A. Tripathi, O. Weiss, A. Wehe, Y. Xu, and M. Yahya, "UPC run-time error detection test suite," 2008. [Online]. Available: http://rted.public.iastate.edu/UPC/

[7] J. Träff and J. Worringen, "Verifying collective MPI calls," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2004, vol. 3241, pp. 95–107, 10.1007/978-3-540-30218-6_11. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30218-6_11

[8] C. Falzone, A. Chan, E. Lusk, and W. Gropp, "Collective error detection for MPI collective operations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, B. Di Martino, D. Kranzlmller, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2005, vol. 3666, pp. 138–147, 10.1007/11557265_21. [Online]. Available: http://dx.doi.org/10.1007/11557265_21

[9] W. Gropp, "Runtime checking of datatype signatures in MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Dongarra, P. Kacsuk, and N. Podhorszki, Eds. Springer Berlin / Heidelberg, 2000, vol. 1908, pp. 160–167, 10.1007/3-540-45255-9_24. [Online]. Available: http://dx.doi.org/10.1007/3-540-45255-9_24

[10] S. Siegel, “Verifying parallel programs with MPI-Spin,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, F. Cappello, T. Herault, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2007, vol. 4757, pp. 13–14, 10.1007/978-3-540-75416-9_8. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75416-9_8

[11] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, “ISP: a tool for model checking mpi programs,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP ’08. New York, NY, USA: ACM, 2008, pp. 285–286. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345258

[12] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, “MODIST: transparent model checking of unmodified distributed systems,” in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, ser. NSDI’09. Berkeley, CA, USA: USENIX Association, 2009, pp. 213–228. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558992

[13] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby, “Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings,” in *Proceedings of the 20th international conference on Computer Aided Verification*, ser. CAV ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 66–79. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70545-1_9

[14] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. de Supinski, M. Schulz, and G. Bronevetsky, “A scalable and distributed dynamic formal verifier for MPI programs,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, nov. 2010, pp. 1 –10.

[15] B. Krammer, M. Müller, and M. Resch, “MPI application development using the analysis tool MARMOT,” in *Computational Science - ICCS 2004*, ser. Lecture Notes in Computer Science, M. Bubak, G. van Albada, P. Sloot, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2004, vol. 3038, pp. 464–471, 10.1007/978-3-540-24688-6_61. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24688-6_61

[16] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva, “Deadlock detection in MPI programs,” *Concurrency and Computation: Practice and Experience*, vol. 14, no. 11, pp. 911–932, 2002. [Online]. Available: http://dx.doi.org/10.1002/cpe.701

[17] J. S. Vetter and B. R. de Supinski, “Dynamic software testing of MPI applications with Umpire,” in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing ’00. Washington, DC, USA: IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=370049.370462

[18] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, “A graph based approach for MPI deadlock detection,” in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS ’09. New York, NY, USA: ACM, 2009, pp. 296–305. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542319

[19] T. Hilbrich, M. Schulz, B. R. Supinski, and M. S. Müller, “MUST: A scalable approach to runtime error detection in mpi programs,” in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Springer Berlin Heidelberg, 2010, pp. 53–66, 10.1007/978-3-642-11261-4_5. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11261-4_5

[20] The UPC Consortium, “UPC Language Specifications (v1.2),” 2005. [Online]. Available: http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf

[21] “UPC NAS Parallel Benchmarks.” [Online]. Available: http://threads.hpcl.gwu.edu/sites/npb-upc

[22] D. J. Quinlan and et al., “ROSE compiler project.” [Online]. Available: http://www.rosecompiler.org/