2012

UPC-CHECK: A scalable tool for detecting run-time errors in Unified Parallel C

Indranil Roy
Iowa State University

UPC-CHECK: A scalable tool for detecting run-time errors in Unified Parallel C

by

Indranil Roy

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Iowa State University

Ames, Iowa

2012

DEDICATION

I would like to dedicate this thesis to my parents and to my loving sister. Without their support I would not have been able to complete this work.

I would also like to thank my friends, especially my roommate for their love, care and support during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Unified Parallel C (UPC) is a language used to write parallel programs for shared and distributed memory parallel computers. UPC-CHECK is a scalable tool developed to automatically detect argument errors in UPC functions and deadlocks in UPC programs at run-time and issue high quality error messages to help programmers quickly fix those errors. The tool is easy to use and involves merely replacing the compiler command with *upc-check*. The tool uses a novel distributed algorithm for detecting argument and deadlock errors in collective operations. The run-time complexity of the algorithm has been proven to be O(1). The algorithm has been extended to detect deadlocks created involving locks with a run-time complexity of O(T), where T is the number of threads waiting to acquire a lock. Error messages issued by UPC-CHECK were evaluated using the UPC RTED test suite for argument errors in UPC functions and deadlocks. Results of these tests show that the error messages issued by UPC-CHECK for these tests are excellent. The scalability of all the algorithms used was demonstrated using performance-evaluation test programs and the UPC NAS Parallel Benchmarks.

Keywords: UPC, run-time error detection, dynamic distributed deadlock detection, distributed shared memory, partitioned global address space.

CHAPTER 1. GENERAL INTRODUCTION

1.1 Introduction

Unified Parallel C (UPC) [17, 6] is an extension of the C programming language for parallel execution on shared and distributed memory parallel machines. UPC is based on the Partitioned Global Address Space (PGAS) [54] programming model. The PGAS model provides the user the ease of programmability of a shared memory programming paradigm while providing fine-grain control over data layout and operations of the message passing paradigm.

However, extensive studies show that the error detection capability of compilers and run-time environments for the UPC language is poor [25, 30]. In the absence of such error detection capabilities in the system software, a programmer needs to debug the program by manually computing the program. This can be tedious and challenging for high performance programs which can be lengthy, complex and written by a team of people over an extended period of time.

To overcome this limitation, UPC-CHECK was envisioned as a run-time error detection tool which can be used to automatically detect argument errors in UPC functions and deadlocks in UPC programs. UPC-CHECK is highly scalable and incurs very low memory and execution-time overhead. The tool can detect all the argument errors listed in UPC RTED test suite [13]. For deadlock detection, it employs an algorithm [43] whose run-time complexity is proven to be O(1) for detecting all deadlocks involving any UPC collective operations. In case of deadlocks involving acquiring of locks, the algorithm has been extended to maintain a distributed and shared Wait-For-Graph (WFG) and detect deadlocks using the AND model. In this case, the complexity is O(T), where T is the number of threads in the WFG.

On detecting an error, UPC-CHECK issues a high quality error message. The message pin points the kind of error, line number and the name of the file where the error occurred and information which might be helpful to the programmer to fix the error quickly. The error detection capability of the tool and the quality of error messages issued were graded using the UPC-RTED [30] tool. The results show a marked improvement in the error detection and reporting capability over existing run-time systems like Cray, Berkeley, HP and GNU.

1.2 Thesis Organization

The thesis consists of two papers. Chapter 2 contains the first paper which describes the UPC-CHECK tool. The paper first outlines various error detection tools for programs written in various parallel programming environments like MPI, OpenMP and UPC. It continues to provide the overview of the tool and the kind of errors which can be detected using the tool. Finally the results of function, scalability and overhead testing have been presented.

The new deadlock detection algorithm used in the UPC-CHECK tool has been explained in the paper reproduced in Chapter 3. The paper provides an elaborate literature review of the deadlock detection methods for distributed systems, multicore operating systems, databases and high performance computing. The algorithm is first presented and then its correctness and run-time complexity are rigorously proven. Various performance evaluation tests and real-time program like the UPC NAS Parallel Benchmark [7] are then used to demonstrate the low overhead and scalability of the algorithm.

Finally, a summary is presented in Section 4. The summary describes the ease of use of the tool which merely involves replacing the compiler command with the 'upc-check' command. A user's guide for using the tool along with some tutorial examples have been presented in Appendix A. The authors hope that this tool will provide a more productive environment for programmers to quickly develop and debug programs using UPC.

CHAPTER 2. UPC-CHECK: A SCALABLE TOOL FOR DETECTING RUN-TIME ERRORS IN UNIFIED PARALLEL C

Modified from a paper accepted in International Supercomputing Conference (ICS) 2012[7]

James Coyle[1][6], Indranil Roy[1][2][3][6], Marina Kraeva[1][6], and Glenn R. Luecke[4][5][6]

2.1 Abstract

Unified Parallel C (UPC) is a language used to write parallel programs for distributed memory parallel computers. UPC-CHECK (http://hpcgroup.public.iastate.edu/UPC-CHECK/) is a scalable tool developed to automatically detect argument errors in UPC functions and deadlocks in UPC programs at run-time and issue high quality error messages to help programmers quickly fix those errors. The run-time complexity of all detection techniques used are optimal, i.e. O(1) except for deadlocks involving locks where it is theoretically known to be linear in the number of threads. The tool is easy to use, and involves merely replacing the compiler command with *upc-check*. Error messages issued by UPC-CHECK were evaluated using the UPC RTED test suite for argument errors in UPC functions and deadlocks. Results of these tests show that the error messages issued by UPC-CHECK for these tests are excellent.

[1]Primary researcher
[2]Graduate student
[3]Primary author
[4]Graduate advisor
[5]Author for correspondence
[6]Iowa State University's High Performance Computing Group,
Iowa State University, Ames, Iowa 50011, USA.
email: jjc@iastate.edu, iroy@iastate.edu, kraeva@iastate.edu and grl@iastate.edu
[7]This work was supported by the United States Department of Defense & used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory.

2.2 Introduction

The importance of error detection is well documented by Glenn Luecke et al. [31]: “the ability of system software to detect run-time errors and issue messages that help programmers quickly correct these errors is an important productivity criterion for developing and maintaining application programs”. However, studies show that currently the error detection capability of compilers and run-time systems for Unified Parallel C (UPC) [10, 3, 17] is poor [25, 30].

Though tools to detect errors in serial, MPI [50, 20, 26, 32, 15] and OpenMP programs [38, 4] exist, no tools for checking UPC programs existed when this work began. The authors are aware of two other tools published since then, ROSE-CIRM with UPC extensions [39] and UPC-SPIN [16]. UPC-CIRM detects “C style” errors in UPC programs but detects neither deadlock errors nor argument errors in UPC functions. Therefore, it has no overlap with UPC-CHECK. UPC-SPIN does not detect argument errors in UPC functions but can detect deadlock errors. UPC-SPIN uses model checking to create a finite model of the parallel program and then analyzes all possible control paths to search for deadlocks. This leads to a combinatorial explosion in complexity both in time and memory space. The developer of UPC-SPIN states in [16] that UPC-SPIN can only be used for small/moderate sized applications. Results from the UPC-SPIN paper show the exponentially increasing time and memory requirements for the analysis of the UPC NAS Parallel Benchmark (NPB) [7] Conjugate Gradient (CG). The results reported show that the analysis could be completed for the model of NPB CG for a maximum of 4 threads. By design, UPC-CHECK avoids this problem by not computing all possible control paths through the program, but instead focusing only on the current execution. Section 2.4.2 of this paper shows UPC-CHECK to be highly scalable, easily handling 128 threads for the NPB CG and the other NPBs with minimal overhead. In fact, the overhead of UPC-CHECK is so low that many applications could always be run with UPC-CHECK.

UPC-CHECK is an error detection tool which detects argument errors in UPC functions and deadlocks in UPC programs at run-time. The tool is easy to use and merely involves replacing the compiler command with *upc-check* on the command-line or in Makefile(s). To make this tool scalable, a new optimal deadlock detection algorithm has been developed. The execution time complexity for finding deadlocks is $O(T)$ where T is the number of threads. The execution time complexity for finding deadlocks when only UPC collective operations are involved is $O(1)$. This algorithm is described in detail in [43, 42].

Section 2.3 provides an overview of UPC-CHECK including its design, functionality and usage. Section 2.4 describes the function, scalability, overhead and compiler-independence testing of UPC-CHECK. Section 2.6 contains an example illustrating how UPC-CHECK can be used to find and correct a deadlock in a UPC program.

2.3 Overview of UPC-CHECK

If an error is allowed to occur, it may not be possible to report information accurately. Therefore, UPC-CHECK has been designed to detect errors before they occur, while the program is active and in a valid state. This allows the instrumented program to issue a correct high quality error message and then exit gracefully.

For UPC-CHECK, both central manager and distributed error detection techniques were considered. The distributed techniques were chosen over the simplicity of a central manager technique for reasons of scalability and low overhead.

The authors considered using either a source-to-source translator or modifying an existing open source UPC compiler. A source-to-source translator was chosen so that the tool would be compiler and machine independent. The ROSE toolkit [40] developed at Lawrence Livermore National Laboratory was chosen to write the UPC-CHECK source-to-source translator.

An overview of the UPC-CHECK tool can be seen in Figure 2.1. In the first step, the UPC-CHECK *translator* instruments the original UPC files. The instrumented files call UPC-CHECK functions which check for error conditions and record information which may be required to issue good error messages.

UPC-CHECK supplies support files which contain the declarations and definitions of all data structures, enumerations and functions used for error checking and issuing errors. In the second step, the instrumented files are compiled along with the UPC-CHECK support files using the user's native UPC compiler to create an executable.

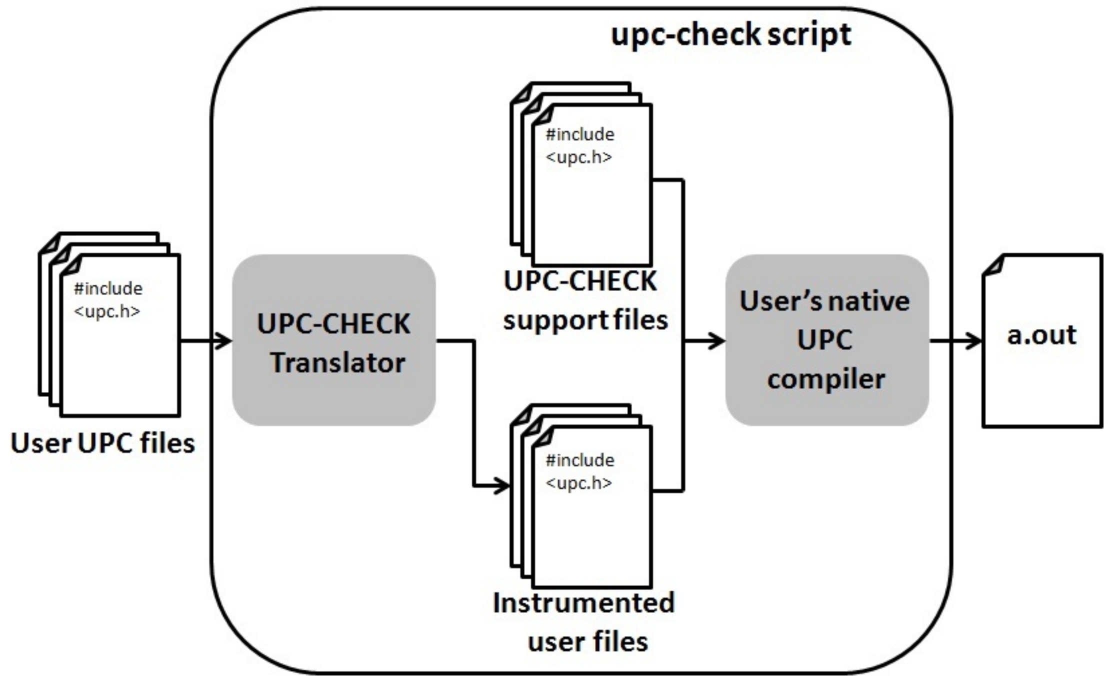

Figure 2.1 Compiling files with UPC-CHECK

UPC-CHECK is designed for ease of use. The user must merely replace the UPC compiler command with `upc-check` either on the command-line or in Makefiles. (UPC-CHECK assumes that the original program compiles with the user's native UPC compiler.)

The generated executable is run in the same manner as those created by using the user's native UPC compiler. However the executable created using UPC-CHECK can detect errors and issue good error messages which can be used to debug the program. Details about the usage of UPC-CHECK can be found in the UPC-CHECK tutorial[29] and user's guide[19].

2.3.1 Instrumentation

Instrumentation is done as follows. Context information is stored in global variables. Information that may be accessed by other threads is stored in shared variables. For execution efficiency, information that is required only within a thread is stored in private variables.

Allocation and necessary initialization of variables, data structures etc. defined by UPC-CHECK are inserted at the beginning of the program. File name and line number information is stored for every UPC operation encountered. The authors define UPC operation to be a UPC function or a UPC statement that is not a C statement. To enforce the UPC specification, instrumentation is inserted to check that no UPC collective routine is called between a upc_notify and an upc_wait. To record that the thread has reached the end of execution, a function call is inserted before every `return` statement in the main function and every `exit()` function.

To check for argument errors in UPC functions, a call to the argument check function is inserted before every UPC function.

To check for deadlock errors, a call is inserted before and after each UPC operation. The call before the operation checks whether executing the operation would cause a deadlock. The call after the operation is to record that the operation is complete and record any additional information that might have been returned. In addition a function call to check for possible deadlock conditions is inserted before every `return` statement in the main function and every `exit()` function. More details of these functions are provided in Section2.3.2.

When tracking of function-call-stack is enabled, calls to functions to update the function-call-stack are inserted before and after calls to user functions.

2.3.2 Errors detected by UPC-CHECK

UPC-CHECK detects all argument errors in UPC functions other than out-of-bound array and pointer accesses and uninitialized variables. UPC-CHECK can detect all deadlocks and livelocks that may arise during the execution of an UPC program.

2.3.2.1 Argument error detection

Before a UPC function call, the argument-error check function determines whether all the arguments which are about to be passed to the UPC function satisfy the conditions set by the UPC specification. There are over 350 argument error checks. These errors can be classified into

- *invalid argument* errors and

- *non-single-valued argument* errors.

Invalid argument errors are mostly related to undefined usage, e.g. passing of a negative number for the thread index, passing undefined flags, etc. They also include error cases where the value passed is inconsistent with values previously defined in the program. An example of such an error is when the thread index passed is larger than the total number of threads used in the program.

Some arguments of UPC collective operations are called single valued arguments. These must have the same value on every thread. When this is not the case, the authors call this a *non-single-valued* argument error.

2.3.2.2 Deadlock error detection

The authors have developed a new scalable algorithm for deadlock detection for UPC programs. The algorithm has a complexity of O(1) when detecting deadlocks except those involving chain of hold-and-wait lock dependencies where it is known to be O(T), where T is the number of threads involved in the hold-and-wait chain. In [43, 42], the authors describe the algorithm in detail, prove its correctness, determine its run-time complexity and prove its optimality.

Detecting deadlock errors caused only by improper usage of collective operations First consider deadlocks that can be created using only collective operations. According to the UPC specification, *collective* operations must be called on every thread and the order of the calls to collective operations must be the same on all threads. Thus, there are two types of deadlocks that could be caused by collective operations violating the above rules:

1. Some threads are waiting at a collective operation while others have finished execution,
2. Different threads are waiting at different collective operations.

Detecting deadlock conditions with locks Acquiring a lock through the upc_lock command is a blocking operation. This can give rise to the well-known circular hold-and-wait deadlock condition for acquiring locks. This is illustrated in Figure 2.2. The boxes in the figure depict threads whereas the circles depict locks. A dashed arrow from a thread to a lock shows that the thread is waiting to acquire the lock at the head of the arrow. On the other hand a solid arrow from the lock to a thread shows that that lock is held by the thread at the head of the arrow.

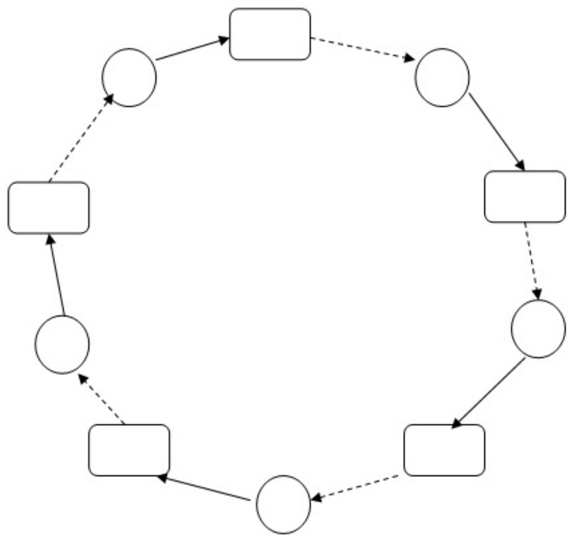

Figure 2.2 Circular dependencies of threads leading to a deadlock.

Secondly, deadlocks could be created if a lock was acquired but was never released by a thread which has completed execution. Similar to the above case, if there is a chain of hold-and-wait dependencies which can be resolved by unlocking this lock then all the threads in that chain would be deadlocked.

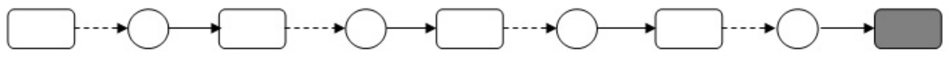

Figure 2.3 Chain of dependencies leading to a thread that is either waiting at a collective operation or has completed execution.

Thirdly, deadlocks could also be created when there is a chain of hold-and-wait dependencies that can be resolved by unlocking a lock which is held by a thread blocked at a collective operation, see Figure 2.3. The boxes in the figure depict a thread whereas the circles depict locks. A dashed arrow from a thread to a lock shows that the thread is waiting to acquire the lock at the head of the arrow. On the other hand a solid arrow from the lock to a thread shows that this lock is held by the thread at the head of the arrow. The thread which is blocked at a collective operation is depicted by the gray box.

The UPC specification places an additional constraint on the use of lock functions. If the upc_unlock(L) function is not called by the thread that holds the lock L, the result is not defined.

Similar to deadlocks, execution of threads does not progress when threads are busy-waiting at a *livelock*. In UPC programs, a livelock can be created when the `upc_lock_attempt` function is called within an infinite loop to acquire a lock which will never be released. UPC-CHECK prints out a warning when a livelock condition is detected and exits after a timeout if set by the user.

2.3.2.3 Handling upc_forall statements

UPC-CHECK detects error conditions that could arise due to illegal control flow into and out of the body of a 'controlling' upc_forall loop. This is achieved by giving an unique index to the code segment belonging to the body of every controlling upc_forall. Any code section outside the body of all controlling upc_forall statements is assigned the index 0. During execution, a variable maintains the index of the code section where the program control lies. The following checks are performed for upc_forall statements and error messages are issued if:

1. any thread encounters a collective operation inside the body of a controlling upc_forall loop,

2. any thread encounters a `return` statement inside the the body of a controlling upc_forall loop or a `break` statement which takes the control outside the body of a controlling upc_forall loop,

3. control jumps to a label inside the body of a controlling upc_forall loop from a goto statement in a region with index different from the region index of the body of this upc_forall loop, and

4. control jumps to a label outside the body of any controlling upc_forall loop from a goto statement in a region with the index of a controlling upc_forall statement.

Checking the single-valuedness of the *condition -expression* and the *affinity-expression* for all the iterations of a upc_forall statement results in serialization of the execution of iterations across all threads. To maintain scalability, checking the single-valuedness of the condition-expression and the affinity-expression of an upc_forall statement has not been implemented in UPC-CHECK.

2.4 Testing

UPC-CHECK has been thoroughly tested not only for known error cases but also against false positive cases. In this section, function, scalability, overhead and compiler-independence testing is decribed.

2.4.1 Function testing

UPC-CHECK was tested against over 350 test cases in the UPC RTED test suite [13]. The test cases in the test suite are categorized under various error categories. Sections F, K.2 and K.3 of this test-suite are related to argument errors in UPC functions, whereas Section B is related to deadlock errors. To evaluate the run-time error detection (RTED) capabilities of UPC run-time systems [30], the High Performance Computing (HPC) Group at Iowa State University (ISU) [2] devised a scheme to score the quality of error messages issued. The scores range from 0 to 5, where 0 means that the error was not detected and 5 means that the error was detected and that the error message contains all the information required to fix the error quickly. The score assigned to each error category is the avergae of the scores of all tests in that category. The scores for argument errors in UPC functions and deadlock categories are presented in Table 2.1.

Table 2.1 illustrates the improvement in the quality of error detection and error messages generated by UPC-CHECK compared to other UPC run-time systems. The scores for UPC-CHECK were found to be consistent across different compilers and machines. A more in-depth

UPC Error Category	**Cray**	**Berkeley**	**HP**	**GNU**	**UPC-CHECK**
Argument errors in UPC functions	0.38	0.17	0.00	0.00	4.89
Deadlocks	0.00	0.33	0.36	0.27	5.00

Table 2.1 Error detection and reporting scores: UPC-CHECK compared to other systems

analysis of the scores reveals that UPC-CHECK achieves a score of 5 in every error test case except for three argument error cases where it does not detect the error. These three argument error cases relate to single-valuedness of the *condition* and *affinity* expressions of the upc_forall statement. This is not checked for in UPC-CHECK as has already been discussed in Section 2.3.2.3.

To further test UPC-CHECK's ability to detect all cases of deadlock conditions that can arise while executing an UPC program, additional tests were written. These deadlock conditions have been discussed in detail in [43, 42].

For every error test, a positive test was created by correcting the error in the original test. UPC-CHECK was tested against these positive tests to verify that no error messages were issued. To demonstrate that UPC-CHECK can be used for larger UPC programs, the UPC NPBs were also run using UPC-CHECK.

2.4.2 Scalability and overhead testing

2.4.2.1 Execution overhead

UPC-CHECK uses a scalable distributed deadlock detection algorithm. The algorithm has a run-time complexity of O(1) in terms of private and shared memory accesses while detecting errors created by UPC collective operations only; and a complexity of $O(T)$ where T is the number of threads when detecting deadlock conditions involving all UPC blocking operations including locks. Checking argument errors in UPC functions has a complexity of O(1) for all UPC functions. To demonstrate the low overhead, an instrumented version was created for each UPC NPB using UPC-CHECK. The execution times of the original and the instrumented versions were compared when run on a CRAY XT with 128 threads. The results are shown in Table 2.2. Execution times where the original benchmark did not run on the machine are marked as Not Applicable(NA). The maximum slowdown is 5.2% and the average slowdown is 0.86%. A slow-down of less than 0 means that the overhead involved due to the use of UPC-CHECK was insignificant and likely caused by timer resolution issues.

Name-Class	**Original (secs)**	**Instrumented (secs)**	**Slow-down (%)**
CG-S	4.742	4.942	4.2
CG-W	15.664	15.708	0.3
CG-A	4.912	4.99	1.6
CG-B	54.183	54.239	0.1
CG-C	58.309	58.281	0
EP-S	1.145	1.145	0
EP-W	6.247	6.243	-0.01
EP-A	1.417	1.427	0.7
EP-B	7.116	7.128	0.2
EP-C	11.19	11.17	-0.2
FT-S	NA	NA	NA
FT-W	NA	NA	NA
FT-A	NA	NA	NA
FT-B	15.528	15.556	0.2
FT-C	22.855	22.735	-0.2
IS-S	3.541	3.594	1.5
IS-W	10.422	10.961	5.2
IS-A	3.56	3.658	2.8
IS-B	8.752	8.776	0.3
IS-C	10.089	10.073	-0.2
MG-S	NA	NA	NA
MG-W	8.288	8.308	0.2
MG-A	NA	NA	NA
MG-B	9.293	9.341	0.5
MG-C	13.551	13.579	0.2
Maximum			5.2
Average			0.86

Table 2.2 Percentage slow-down of various UPC NAS parallel benchmark on 128 threads.

2.4.2.2 Memory Overhead

The memory requirement per thread of a program instrumented with UPC-CHECK is less than 128 kilobytes per thread. If enabled, call stack tracking adds 0.5 kilobytes times the maximum call depth. On average for every UPC function, 100 lines are added in the instrumented program. Additionally, the support files add about 12,000 lines to the program.

2.5 Compiler-independence testing

UPC-CHECK was tested for compiler-independence using all function tests in Section 2.4.1 with the CRAY, Berkeley and GNU UPC compilers. All tests ran and produced identical error messages for the three compilers.

The instrumented files and the support files are regular UPC files and thus can be compiled by any compiler which conforms to the UPC specification. For UPC compilers that do not support UPC-IO and/or UPC collectives, UPC-CHECK can still be used by setting appropriate environment variables as described in the user's guide[19].

2.6 Fixing errors using UPC-CHECK: An example

In this section, the authors present a simple example which shows how UPC-CHECK can be used to quickly fix an error in a program. The program consists of two files ex3.upc and ex3_s.upc and has a deadlock error condition because the `upc_barrier` function is not called by all threads. This error is difficult to find since the barrier is contained inside a function which is called from within an `if` block. When issuing:

```
upcc -T 4 -o ex3 ex3.upc ex3_s.upc
upcrun -n 4 ./ex3
```

a deadlock occurs and the upcrun command never returns. When issuing:

```
upc-check -T 4 -o ex3 ex3.upc ex3_s.upc
upcrun -n 4 ./ex3
```

the following message is issued:

```
Runtime error: Deadlock condition detected: One or more threads have
finished executing while other threads are waiting at a collective routine
Status of threads
=================
Thread id:Status:Presently waiting at line number:of file

0:waiting at upc_barrier: 7: /home/jjc/ex3_s.upc
1:reached end of execution through: 39: /home/jjc/ex3.upc
2:waiting at upc_barrier: 7: /home/jjc/ex3_s.upc
3:waiting at upc_barrier: 7: /home/jjc/ex3_s.upc
```

Using this error message, the error can be quickly identified and fixed. The `upc_barrier` is called from *funcA*. Two of the three possible paths through the two nested `if` statements appear and contain a `upc_barrier`, but the third possible (`else`) path is missing. This error can be corrected by creating the missing `else` block at line 25 and placing either a call to *funcA*, or a `upc_barri`
`-er` call. If the location of the calls to *funcA* were not obvious, UPC-CHECK call-stack tracing could be enabled to provide that information.

ex3.upc

```
...
6 /* function that returns an integer zero value is unlikely to be computed
at compile time */
7 int zero(){
8      return (int) (sin(0.1*MYTHREAD)/2.3);
9 }
...
20      /* called by thread 1 only */
21      if (MYTHREAD == 1) {
22          if (zero()) {
23              funcA();
24          }
25          /* Missing else block */
26      }
27
28      /* called by all other threads */
29      else {
30          funcA();
31      }
...
```

```
ex3_s.upc
...
void funcA() {
      upc_barrier;
}
...
```

2.7 Summary

UPC is a language used to write parallel programs for distributed memory parallel computers. UPC compilers and run-time systems currently exhibit poor error detection capabilities. UPC-CHECK is a tool developed to automatically detect argument errors in UPC functions and deadlocks in UPC programs at run-time and issue high quality error messages to help programmers quickly fix those errors.

UPC-CHECK uses a new scalable algorithm for deadlock detection. The average run-time overhead of running UPC-CHECK on UPC NAS parallel benchmarks was less than 1% for 128 threads. UPC-CHECK has been extensively tested with over five hundred error test programs using CRAY, Berkeley and GNU UPC compilers. Error messages issued by UPC-CHECK were evaluated using the UPC RTED test suite [13] for argument errors in UPC functions and deadlocks. Results of this testing show that the error messages issued by UPC-CHECK for these tests are excellent.

UPC-CHECK has been designed for ease of use. It comes with a script to install itself and all software upon which it is dependent. Using UPC-CHECK involves merely replacing the compiler command with *upc-check* on the command-line or in Makefile(s). Necessary documentation in the form of a user's guide and tutorial is available.

CHAPTER 3. A DISTRIBUTED, SCALABLE AND OPTIMAL DEADLOCK DETECTION ALGORITHM FOR UNIFIED PARALLEL C

Modified from a paper to be submitted to International Conference for High Performance Computing, Networking, Storage and Analysis (SC) 2012[7]

Indranil Roy[1236], Glenn R. Luecke[456], James Coyle[16], Marina Kraeva[16], and

3.1 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 deadlock scenarios that involve *collective* operations along with resource deadlocks defined in the popular AND resource-request model. In this paper, a distributed and scalable deadlock detection algorithm for UPC programs that uses run-time analysis is presented. The algorithm detects deadlock scenarios in collective operations using a distributed technique with O(1) run-time complexity irrespective of the collective operations involved. It combines this technique with detecting resource deadlocks that involve acquiring locks by identifying cycles in a shared wait-for-graph (WFG). The correctness and the optimality of the algorithm has been proven. 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

[1]Primary researcher
[2]Graduate student
[3]Primary author
[4]Graduate advisor
[5]Author for correspondence
[6]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 and kraeva@iastate.edu

detection algorithm has been experimentally verified.

3.2 Introduction

Unified Parallel C (UPC) is an extension of the C programming language for parallel execution on shared and distributed memory parallel machines [17, 6]. It uses the Partitioned Global Address Space (PGAS) [54] parallel programming model where shared variables may be directly read and written by any thread. Shared variables that are stored in the memory of the thread are said to have *affinity* to that thread. “UPC combines the programmability advantages of the shared memory programming paradigm and the control over data layout and performance of the message passing programming paradigm” [5].

In this paper, a new deadlock detection algorithm to check for deadlocks in UPC programs is presented. The algorithm uses run-time analysis. Its run-time complexity is proven to be O(1) for detecting all deadlocks involving any UPC collective operations. In case of deadlocks involving acquiring of locks, the algorithm has been extended to maintain a distributed and shared Wait-For-Graph (WFG) and detect deadlocks using the AND model. In this case, the complexity is O(T), where T is the number of threads in the WFG. Since the information must be obtained from all threads in the deadlock, this method must be optimal. This deadlock detection method has been implemented as part of the run-time error detection tool UPC-CHECK [14]. The tool sucessfully detects all deadlock error test cases from the UPC RTED test suite [13]. Scalability of this deadlock detection algorithm has been experimentally verified.

The rest of this paper is organized as follows. Section 3.3 provides the background of different deadlock detection techniques used in distributed systems, multi-core operating systems and high performance parallel cluster machines. In Section 3.4, all deadlock and livelock conditions which might arise in a UPC program are presented. An algorithm to detect these deadlock conditions is described. The correctness, run-time complexity analysis and proof of optimality of the algorithm are also provided. In Section 3.5 the scalability of the implementation of the algorithm has been presented and analyzed.

3.3 Background

Detecting deadlocks in various multi-process systems is well-studied. Depending on the type of underlying resource-request model [24], deadlocks can occur in different ways. The most generalized model in distributed systems is called the *N-out-of-M* model [8]. In this model, if N out of the M resources requested by a process are granted, then the process is no longer blocked. Two common special cases of this generalized model are: the AND (resource) model where a process is blocked until it is granted all the resources it has requested, i.e. when N=M; and the OR model(communication) model where a process is blocked until it is granted any one of the resources that it has requested, i.e. when N=1. There exist several algorithms to detect deadlocks in the *N-out-of-M* model [8, 27, 11, 28], the AND model [9, 12, 18, 37, 41, 46, 52, 44, 23, 16, 50, 21] and the OR models [9, 22, 33, 36]. These algorithms maintain a dependency graph called either a *wait-for graph* (WFG) or a resource allocation graph (RAG) based on resources requested and resources acquired by different processes in the system. In WFG, nodes represent blocked processes that are waiting to acquire a resource and a directed edge (u,v) symbolizes a resource requested by process u which has been acquired and has still not been released by process v. A cycle in the WFG represents a deadlock in the AND model. Similarly, a *knot* in the WFG represents a deadlock in the OR model. A node u in the WFG is defined to be in a *knot* if all nodes reachable from u can reach u. Algorithms to detect deadlocks in distributed systems adopt centralized [28, 18, 16, 50, 21], distributed [8, 27, 9, 37] or hybrid [11] methods to maintain WFGs or images of WFGs. In a multiprocessor shared memory environment, the WFG can be stored in shared memory [52, 44, 23]. An alternative to using WFGs to find cycles of dependencies is to use *edge-chasing* methods [35, 34, 46, 12, 41]. In these methods, a process u waiting for resources sends a probe message to all processes that the process depends on. If there is a circular dependency, then u recieves the probe message sent by itself which indicates the presence of a deadlock.

Detecting deadlocks in parallel programs for high performance computing (HPC) may need to cover two sources of non-determinism: (a) conditional control-flow and (b) the semantics of blocking communication operations. Deadlock detection techniques in HPC can be classified based on the amount of non-determinism that the techniques cover. The higher the amount of non-determinism covered, the better is the formal verification of the program and lower is the scalability. Deadlock detection techniques in HPC arranged in descending order of coverage of non-determinism are model-checking, dynamic formal analysis and run-time analysis. In model-checking [45, 16] a finite model of the program is created which simulates the dependencies in the original program and then the model is checked for all possible deadlocks in all possible execution paths. Therefore the model-checkers cover the non-determinism of control-flows as well as non-determinism of the communication operations. Unfortunately, this solution may not scale well for real-world problems. Dynamic formal verification methods [49, 53, 48] do not cover the non-determinism of the control flow and hence can be used for larger programs. These methods intercept operations of interest and check for all deadlock conditions covering the non-determinism of the operation. Such methods generally employ centralized deadlock detection schemes which limits them to verifying executions using a small number of processes. Due to poor utilization of cluster-resources, execution time of such methods is usually very high. DAMPI [51] is a dynamic formal verfication tool which overcomes this shortcoming by using a distributed deadlock detection algorithm based on Lamport's clocks only for non-deterministic MPI operations and by using heuristics to bound the non-determinism of the MPI operations. Run-time analysis is the least strict method of testing since it covers neither the non-determinism of the control flow nor the non-determinism of the operation. Such methods only cover the execution path followed by the program during that specific run and the *result* of the non-determinism of the operation in that particular execution. Run-time analysis methods may employ synchronized time-out based strategies [26, 32] or may create a central process which maintains a WFG [26, 21] to detect deadlock conditions. It is noteworthy that WFGs for HPC programs like MPI programs [20] need to be different from the ones explained for AND and OR models. This is because *collective* operations in such languages create dependencies on a set of processes.

When compared to MPI, deadlock scenarios in UPC is marginally simplified. Firstly, communication between two processes is non-blocking and secondly, non-determinism of collective operations in terms of 'any_source' cannot occur. However, restrictions still exist on the order of collective operations executed by each thread and the values passed to the *single-valued* arguments passed on each thread. Non-adherence to these restrictions could lead to a deadlock.

3.4 Methodology

Our algorithm for deadlock detection in UPC programs uses run-time analysis . The algorithm detects all possible deadlock conditions, including:

1. errors in collective operations and
2. unsatisfiable hold-and-wait dependencies when acquiring locks

The algorithm is implemented by inserting a call to:

- *check_entry()* function before each UPC operation which checks whether executing the operation would cause a deadlock,
- *record_exit()* function after each UPC operation to record that the operation is complete and record any additional information that might have been returned, and
- *check_final()* function before every `return` statement in the `main()` function and every `exit()` function to check for possible deadlock conditions.

Inserting these function calls can be achieved using a source-to-source translator. The implementation details have been left out in this paper. Interested readers are referred to [14].

Some 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. the *state* of a thread is defined as the enumeration of the UPC operation that a 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 the enumeration `unknown`.

3. a *single-valued* argument is an argument of a UPC collective operation which must be passed the same value on every thread.

4. the *signature* of a UPC operation on a thread means the enumeration of the UPC operation and the values which are about to be passed to each of the single-valued arguments of the UPC operation on that thread.

3.4.1 Detecting deadlocks due to collective errors in collective operations

The UPC specification requires that the sequence of calls to UPC collective operations must be the same across all threads [47]. Additionally, each '*single-valued*' argument of a collective operation must have the same value on all threads. Therefore while using collective UPC operations, deadlocks could 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 some have finished execution.

An algorithm to check whether any of the above 3 cases is going to occur needs to compare the collective operation which the threads are going to execute next and their single-valued arguments. 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. 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 has been illustrated in Figure 3.1. The correctness of this approach is proven in Section 3.4.1.3.

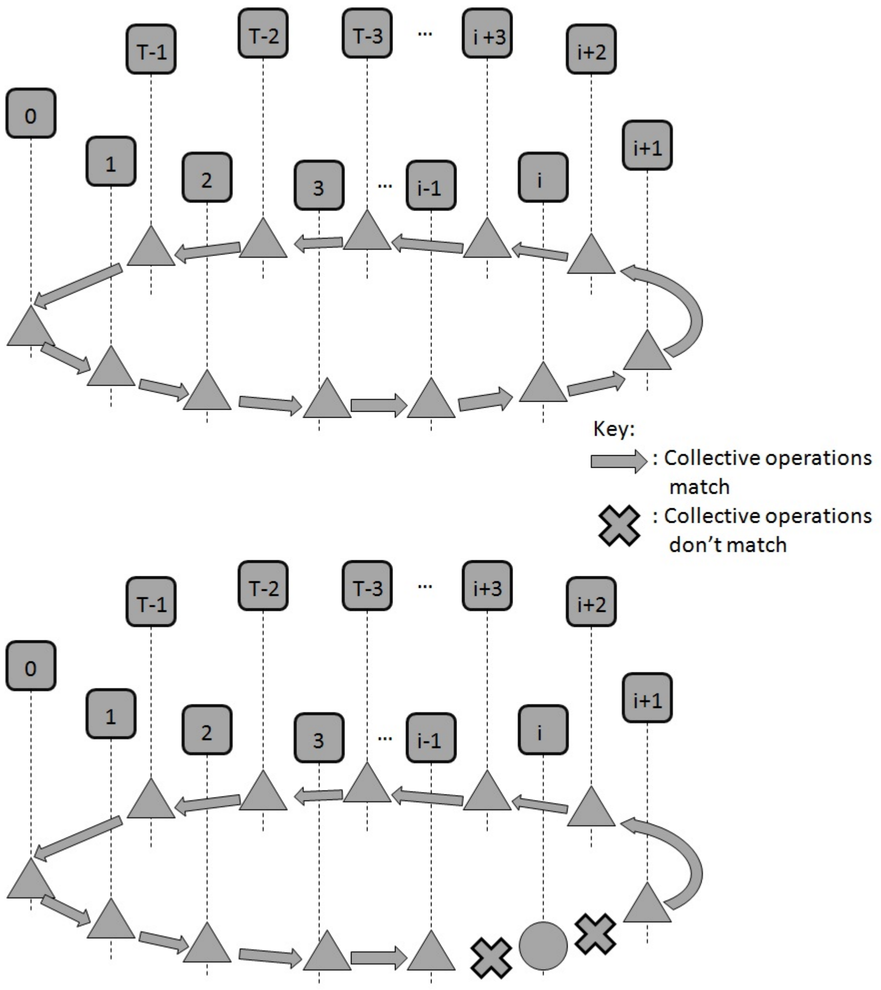
0
1
2
3
...
i-1
i
i+1
i+2
i +3
T-3
T-2
T-1
Key:
: Collective operations match
: Collective operations don't match
0
1
2
3
...
i-1
i
i+1
i+2
i+3
T-3
T-2
T-1

Figure 3.1 Creating consensus: threads checking *state* in a circular ring fashion.

For any thread k, s_k is a shared data structure which stores the *signature* of the collective operation that thread has reached. The field $s_k.op$ stores the *state* of thread k. On reaching a 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 collective operation.

Let $C(n, k)$ denote the signature of the n^{th} collective operation executed by thread k. Also, let thread j be the right neighbor of thread i. During execution, thread i or thread j could reach their repective n^{th} collective operation first. If thread i reaches the operation first, then it cannot verify s_j as s_j has still not be assigned the value of $C(n, j)$. The verification can

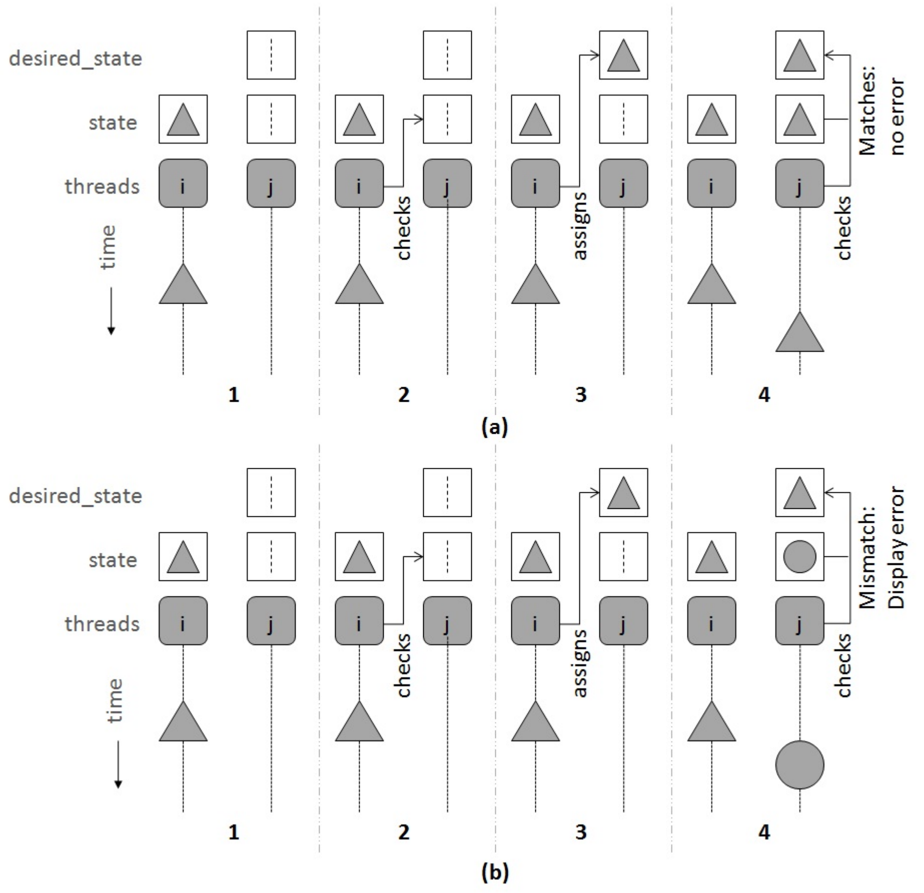

Figure 3.2 Checking *state*: Thread i reaches collective operation before thread j. (a) no error case. (b) error case.

be delayed until thread j reaches its n^{th} collective operation. In order to implement this, we introduce another state-machine variable named *desired_signature*. The *desired_signature* of any thread k is represented by a shared variable named ds_k. Both s_k and ds_k have *affinity* to thread k. If thread i finds that thread j has not reached a collective operation ($s_j.op$ is `unknown`), then it copies 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 has been illustrated in Figure 3.2.

If thread i reaches the collective operation after thread j ($s_j.op$ is set to the enumeration of some collective UPC operation), then thread i compares s_j with s_i. If they match, then there

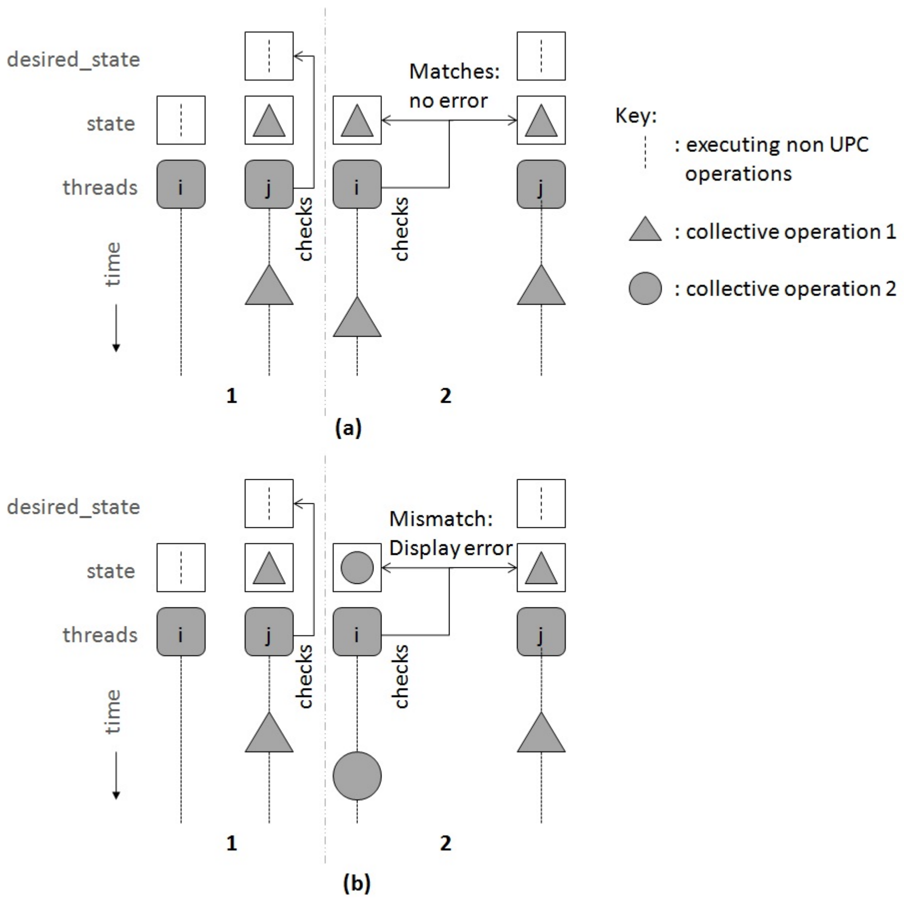

Figure 3.3 Checking *state*: Thread i reaches collective operation after thread j. (a) no error case. (b) error case.

is no error, so execution continues. This has been illustrated in Figure 3.3.

In UPC, collective operations may or may not be synchronizing. To ensure that for neighboring threads i and j, $C(n,i)$ is compared to $C(n,j)$ we have to ensure that:

1. If thread i reaches the n^{th} collective operation before thread j and sets ds_j to $C(n,i)$, it does not rewrite it 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 sets s_j to $C(n,j)$, it does not rewrite 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 rewriting s_j before the comparisons described above. Similarly, variable r_ds_j is used to prevent thread i from rewriting ds_j before the above comparisons are made. Both r_s_j and r_ds_j are stored in the affinity of 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 $L[j]$.

The pseudo-code of the distributed algorithm on each thread i to check deadlocks caused by incorrect or missing calls to collective operations has been presented below. While checking whether the thread has reached the right collective operation, both the enumeration of the collective operation and its arguments are checked. For conciseness, the following operations are defined:

1. $b_j \leftarrow a_i$ means
   (a) assign value of variable $a_i.op$ to variable $b_j.op$, and
   (b) if $a_i.op \neq end\_of\_execution$, copy values of single-valued arguments from thread i to shared space on thread j
2. $b_j \ncong a_i$ is true if
   (a) $b_j.op \neq a_i.op$, or
   (b) if $a_i.op \neq end\_of\_execution$, any of the single-valued arguments on thread j is not identical to its corresponding argument on thread i

Function *check_entry*() recieves as argument the signature of the collective operation that the thread has reached, namely f_sig.

3.4.1.1 Algorithm A1: Detecting unmatched sequences of calls to collective operations

```
1: On thread $i$:
```

```
2:
3: Initialization
4: $s_i.op \leftarrow ds_i.op \leftarrow unknown$, $r\_s_i \leftarrow 1$, $r\_ds_j \leftarrow 1$
5:
6: {Function definition of check_entry(f_sig):}
7: if $THREADS = 1$ then
8:   Exit check.
9: else
10:
11:   Acquire $L[i]$
12:   $s_i \leftarrow f\_sig$
13:   $r\_s_i \leftarrow 0$
14:   if $ds_i.op \neq unknown$ then
15:     {Thread $i-1$ reached the collective operation before thread $i$}
16:     if $ds_i \ncong s_i$ then
17:       Print error and call global exit function.
18:     end if
19:     $r\_s_i \leftarrow 1$
20:     $r\_ds_i \leftarrow 1$
21:     $ds_i.op \leftarrow unknown$
22:   end if
23:   Release $L[i]$
24:   Wait until $r\_ds_j = 1$
25:   Acquire $L[j]$
26:   {This thread is supposed to check $s_j$ or set $ds_j$}
27:   if $s_j.op = unknown$ then
28:     {Thread $j$ hasn't reached the collective operation. Set the $ds_j$, for thread
        $j$ to check against when it reaches a collective operation.}
```

```
29:     $ds_j \leftarrow s_i$
30:     $r\_ds_j \leftarrow 0$
31:   else
32:     {Check if thread $j$ reached the same collective operation}
33:     if $s_j \ncong s_i$ then
34:       Print error and call global exit function
35:     end if
36:     $r\_s_j \leftarrow 1$
37:   end if
38:   Release $L[j]$
39: end if
40: ————————————————————————————————
41: {Function definition of check_exit():}
42: Wait until $r\_s_i = 1$
43: Acquire $L[i]$
44: $s_i.op \leftarrow unknown$
45: Release $L[i]$
46: ————————————————————————————————
47: {Function definition of check_final():}
48: Acquire $L[i]$
49: $s_i.op \leftarrow end\_of\_execution$
50: Release $L[i]$
51: ————————————————————————————————
```

3.4.1.2 Detecting additional 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 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).

It is noteworthy that the above checks mandate that 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 ordering of upc_wait across the threads. The ordering of the upc_wait statements across the threads is automatically guaranteed with the above mentioned checks. This is reflected in Algorithm $A2$.

3.4.1.3 Proof of Correctness

The proof of correctness is structured as follows. First, it is proved that the algorithm is free of deadlocks and livelocks. Second, Lemma 3.4.1 and Lemma 3.4.2 are used to prove that for any thread j, $C(n, j)$ is compared to $C(n, i)$ in Lemma 3.4.3. Finally, using Lemma 3.4.3, 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 definitely issued if even one thread has reached a collective operation with a signature different from any other thread. Case 1 is proved by Theorem 3.4.4 and Case 2 is proved by Theorem 3.4.5.

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, any thread must eventually exit the *while* loops on line 24 and 42. On reaching $C(n,i)$, 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 end of execution or an UPC collective operation. In the former case, a deadlock condition is detected and an error message is issued. 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, it would definitely exit the waiting 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 due to the atomic nature of operations accorded by $L[j]$. When thread i finally acquires $L[j]$, the conditional statement on line 27 mast evaluate to false. If thread i has reached a collective operation with a signature different for $C(n,j)$ 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 3.4.1 *After thread i assigns $C(n,i)$ to ds_j, then thread i does not rewrite 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 rewrite 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 3.4.2 *After thread j assigns $C(n,j)$ to s_j, then thread j does not rewrite s_j before it is compared with s_i.*

Proof After thread j assigns $C(n,j)$ 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 3.4.3 *For any neighboring threads i and j, $C(n,i)$ is always compared with $C(n,j)$.*

Proof This is proved through induction.

Basis. $C(1,i)$ is compared with $C(1,j)$. If thread i reaches collective operation $C(1,i)$ first, then it sets ds_j to $C(1,i)$. Using Lemma 3.4.1, thread i cannot modify 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 3.4.2 states that after thread j assigns $C(1,j)$ to s_j, thread j does not rewrite s_j before it is compared with s_i. The comparison, with s_i is done by thread i after it reaches its first collective operation and has set s_i to $C(1,i)$.

Inductive step. If $C(n,i)$ is compared with $C(n,j)$, then it can be proven that $C(n+1,i)$ is compared with $C(n+1,j)$. If thread i reaches its next collective operation $C(n+1,i)$ first, then it sets ds_j to $C(n+1,i)$. Using Lemma 3.4.1, thread i cannot modify 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 3.4.2 states that after thread j assigns $C(n+1,j)$ to s_j, thread j does not rewrite s_j before it is compared with s_i. The comparison, with s_i is done by thread i after it reaches its next collective operation and set s_i to $C(n+1,i)$.

Using Lemma 3.4.3, it is proven that for any neighboring thread pair i and j, the n^{th} collective operation of thread i is compared with the 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, the n^{th} collective operation encountered by each thread is considered. 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 recording the signature of the encountered collective operation, i.e. line $s_i \leftarrow f\_sig$, notice that for thread i:

s_i must be $C(n, i)$,

ds_i must be either U or $C(n, i-1)$,

s_j must be either U or $C(n, j)$, and

ds_j must be U.

Theorem 3.4.4 *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 $s_i \leftarrow f\_sig$, $C(n, i)$ is the same for all i. Let C denote this common signature. We will prove this theorem by contradiction. An error message is printed only if:

1. $ds_i \neq U$ and $ds_i \neq s_i \Rightarrow ds_i = C$ and $ds_i \neq C \Rightarrow C \neq C$ (contradiction) or

2. $s_j \neq U$ and $s_j \neq s_i \Rightarrow s_j = C$ and $s_j \neq C \Rightarrow C \neq C$ (contradiction)

So Theorem 3.4.4 is proved.

Theorem 3.4.5 *If even one thread has reached a collective operation with a signature different from any other thread, then an 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 signature of the collective operations reached on every thread is not identical, there must be some thread i for which $C(n, i) \ncong C(n, j)$.For these threads i and j, the following actions are 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 $s_j \ncong s_i$ 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 $ds_j \ncong s_j$ 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 $ds_j \leftarrow C(n,i)$ and continues. Then in action 2, thread j executes $s_j \leftarrow C(n,j)$, finds that $ds_j \neq U$ and hence computes $ds_j \ncong s_j$. Now, since $ds_j = C(n,i)$ and $s_j = C(n,j)$ and $C(n,i) \neq C(n,j)$ (by assumption) $\Rightarrow ds_j \ncong s_j$ is true. Therefore thread j issues an error message.

In the second case, in action 2, thread j assigns $s_j \leftarrow C(n,j)$, finds $ds_j = U$ and continues. Before thread i initiates action 1 by acquiring $L[j]$, it must have executed $s_i \leftarrow C(n,i)$. If $ds_i \neq U$ and $ds_i \ncong s_i$, then an error message is issued by thread i, otherwise it initiates action 1. Thread i finds $s_j \neq U$ and computes $s_j \ncong s_i$. Now, since $s_i = C(n,i)$ and $s_j = C(n,j)$ and $C(n,i) \ncong C(n,j)$ (by assumption) $\Rightarrow s_j \ncong s_j$ is true. Therefore thread i issues an error messages.

Since the above two cases are exhaustive, an error is always issued if $C(n,i) \ncong C(n,j)$ and hence Theorem 3.4.5 is proved.

3.4.1.4 Complexity analysis

:

The complexity of the Algorithm $A1$ is O(1) and can be proven as follows:

- thread i only acquires lock $L[i]$ and $L[j]$ which can only be held by its left and right neighbors respectively,
- thread i may wait for atmost its left and right neighbor.

Theorem 3.4.6 *Algorithm A1 is optimal.*

Proof . The input of any algorithm which determines whether all the threads issue the same collective operation is Ω(THREADS). Therefore per thread the input complexity is $\Omega(1)$. In other words, for any deterministic algorithm every thread will at least need to let the algorithm know what collective operation it is going to execute. Given that algorithm $A1$ has a complexity of O(1), and any solution to this problem is $\Omega(1)$, algorithm $A1$ must be optimal.

3.4.2 Detecting deadlocks and livelocks 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 amongst two or more 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 3.4. 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.

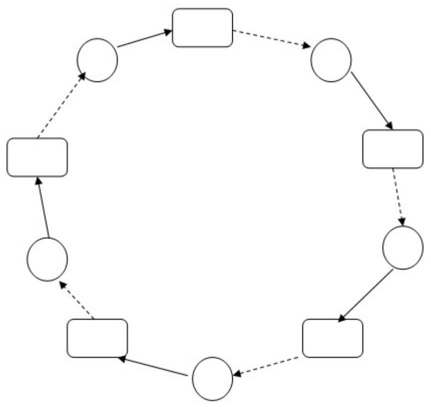

Figure 3.4 Circular dependencies of threads leading to a deadlock.

Using the same notations for locks, threads, hold and wait actions, Figure 3.5 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 grey 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.

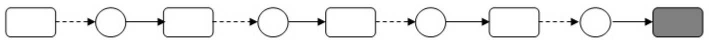

Figure 3.5 Chain of hold-and wait dependencies while trying to acquire a lock leading to a deadlock.

Before a thread l tries to acquire a lock, it checks if the lock is free or not. If it is free, the thread continues exection. Otherwise, if the lock is held by thread q, thread l checks $s_q.op$ to check if thread q:

1. is not executing a collective UPC operation or `upc_lock` operation ($s_q.op$ 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 q is waiting to acquire a lock, then thread l continues to check the *state* of the next thread in the chain of dependencies. If thread l finally reaches thread m which is not executing a collective UPC operation or `upc_lock` operation, then no deadlock is detected. If thread l finds itself along the chain dependencies, then it reports a deadlock condition. Similarly, if thread l finds thread m 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 3.6 illustrates these two cases. Thread l is trying to acquire a lock in a chain of dependencies ending with thread m. When thread l checks the $s_m.op$ of thread m, thread m may (a) not have returned from the synchronizing collective operation $C_s(n, m)$, (b) have returned from $C_s(n, m)$ but not have updated the $s_m.op$ in the $check\_exit()$ function after $C_s(n, m)$, (c) have completed executing $check\_entry()$ function for the next synchronizing collective operation $C_s(n + 1, m)$, or (d) waiting at synchronizing collective operation $C_s(n+1, m)$. $C_s(n, m)$ must be a validated synchronization operation that all threads must have called. 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 l can now identify the scenarios by just comparing $sync\_phase_l$ and $sync\_phase_m$. 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.

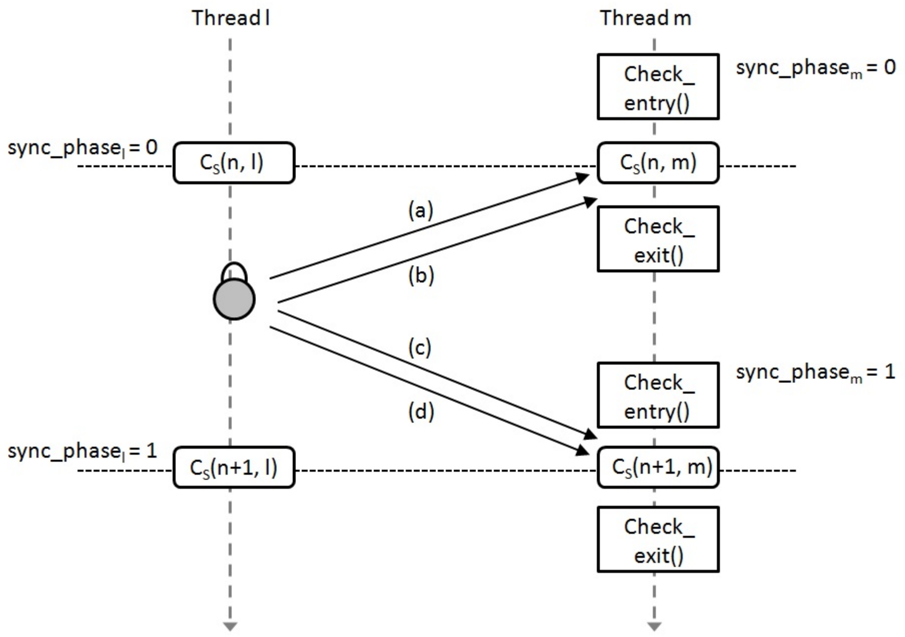

Figure 3.6 Possible scenarios while detecting deadlocks involving chain of hold-and wait dependencies. Scenario (a) or (b) is not a deadlock condition, while scenario (c) or (d) is.

Similar to deadlocks, execution of threads does not progress when threads are busy-waiting at a *livelock*. In UPC programs, a livelock condition is created when *upc_lock_attempt* function is called within an infinite loop to acquire a lock which will not be released. A livelock can be detected using the same WFG and deadlock conditions. Therefore, the same algorithm can be used to identify both deadlocks and livelocks. In case of livelocks, a warning is issued only after the same set of dependencies persists for a user-defined interval of time.

3.4.3 The complete 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 function *check_entry*() recieves two arguments: 1) the signature of the UPC operation that the thread has reached, namely f_sig and 2) the pointer L_ptr which 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. The *check_entry*() and *record_exit*() functions have two arguments. The first argument f_sig is the signature of the UPC operation that the thread has reached. The second argument L_ptr points to a lock if the thread has reached a *upc_lock*, *upc_lock_attempt* or *upc_unlock* statement. *Algorithm A2.*

```
1: On thread $i$:
2: ____________________________________________
3: Initialization
4: Create empty list of acquired and requested locks
5: $s_i.op \leftarrow ds_i.op \leftarrow unknown$, $r\_s_i \leftarrow 1$, $r\_ds_j \leftarrow 1$, $(sync\_phase_i \leftarrow 0)$
6: ____________________________________________
7: {Function definition of check_entry(f_sig, L_ptr):}
8: Acquire $L[i]$
9: $s_i \leftarrow f\_sig$
10: Release $L[i]$
11: if $f\_sig.op = at\_upc\_wait\_statement$ then
```

```
12:     Exit check
13: else if $f\_sig.op = at\_upc\_lock\_operation$ then
14:     Acquire $c\_L$
15:     if $L\_ptr$ is already held by this thread then
16:         Print suitable error and call global exit
17:     else if this dependency is part of a chain leading to a lock which is held by
        a thread finished execution without freeing the lock then
18:         Print suitable error and call global exit
19:     else if this dependency is part of a chain leading to a lock which is held by a
        thread which is blocked at an out-of-phase synchronizing collective operation
        then
20:         Print suitable error and call global exit
21:     else if this dependency creates a circular chain of hold-and-wait dependencies
        then
22:         Print suitable error and call global exit
23:     else
24:         Update list of requested locks
25:         Release $c\_L$
26:         Exit check
27:     end if
28: else if $f\_sig.op = at\_upc\_unlock\_operation$ then
29:     if $L\_ptr$ is not held by this thread then
30:         Print suitable error and call global exit.
31:     else
32:         Update list of acquired locks
33:         Exit check
34:     end if
35: else if $f\_sig.op = at\_upc\_lock\_attempt\_operation$ then
```

```
36:   Acquire $c\_L$
37:   if $L\_ptr$ is already held by this thread then
38:     Print suitable warning
39:   else if this dependency is part of a chain leading to a lock which is held by
      a thread finished execution without freeing the lock then
40:     Print suitable warning
41:   else if this dependency is part of a chain leading to a lock which is held by
      a thread which is blocked at an out-of-phase synchronizing collective routine
      then
42:     Print suitable warning
43:   else if this dependency creates a circular chain of hold-and-wait dependencies
      then
44:     Print suitable warning
45:   else
46:     Update list of requested locks
47:   end if
48:   Release $c\_L$
49:   Exit check
50: else
51:   {Thread must have reached a collective operation}
52:   if $THREADS = 1$ then
53:     Exit check.
54:   end if
55:   Acquire $c\_L$
56:   if this thread holds locks which are in the list of requested locks then
57:     Print suitable error and call global exit.
58:   end if
59:   Release $c\_L$
```

```
60: Acquire $L[i]$
61: $r\_s_i \leftarrow 0$
62: if this is a synchronizing collective operation then
63:   $sync\_phase_i \leftarrow (sync\_phase_i + 1)\%2$
64: end if
65: if $ds_i.op \neq unknown$ then
66:   {Thread $i-1$ reached the collective operation before thread $i$}
67:   if $ds_i \ncong s_i$ then
68:     Print error and call global exit function.
69:   end if
70:   $r\_s_i \leftarrow 1$
71:   $r\_ds_i \leftarrow 1$
72:   $ds_i.op \leftarrow unknown$
73: end if
74: Release lock $L[i]$
75: Wait until $r\_ds_j = 1$
76: Acquire lock $L[j]$
77: {This thread is supposed to check $s_j$ or set $ds_j$}
78: if $s_j.op = unknown$ then
79:   {Thread $j$ hasn't reached the collective operation. Set the $ds_j$, for thread
      $j$ to check against when it reaches a collective operation.}
80:   $ds_j \leftarrow s_i$
81:   $r\_ds_j \leftarrow 0$
82: else
83:   {Check if thread $j$ reached the same collective operation}
84:   if $s_j \ncong s_i$ then
85:     Print error and call global exit function
86:   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 $L[i]$
$s_i \leftarrow end\_of\_execution$
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 time to each thread in the cycle. This means that it adds time of only O(T) where T is the number of threads involved in the dependency chain. This is optimal within a constant as detecting chains and cycles requires information to be obtained from each thread in the chain or cycle. For all other UPC operations, the algorithm is of O(1) complexity which we have already proven to be optimal. Thus our algorithm is optimal for detecting deadlocks in UPC and checking for single-valuedness of arguments of collective UPC operations in applications written using the UPC language.

3.5 Experimental verification of scalability

This deadlock detection algorithm has been implemented in the UPC-CHECK tool [14]. UPC-CHECK was used to experimentally verify the scalability of this algorithm on a Cray XT machine running the CLE 3.1 operating system and the Cray C 7.4.3 compiler. This machine has 20 compute nodes, each node having 8 cores making a total of 160 cores. Since we are interested in the verification of scalability, the authors measured the overhead of our deadlock detection method for 16, 32, 64 and 128 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 [7].

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 549 to 572 microseconds for 16, 32, 64 and 128 threads. When replacing upc_all_broadcast with upc_all_gather_all, overhead times ranged from 512 to 528 microseconds. In both cases, the overhead is nearly constant and confirms that the overhead of deadlock checking does not depend on the number of threads, i.e. the run-time complexity of the deadlock detection algorithm for collective operations is O(1).

Timing results for the UPC NPB CG and IS benchmarks are presented in Tables 3.1 and 3.2. These results also demonstrate the scalability of the deadlock detection algorithm presented in this paper.

3.6 Conclusion

In this paper, a new distributed, optimal and scalable deadlock detection algorithm using run-time analysis has been presented for UPC programs. The algorithm utilizes a distributed

Threads	Class B			Class C		
	Without checks	With checks	Overhead	Without checks	With checks	Overhead
16	15.4	29.8	14.4	34.0	61.6	27.6
32	14.1	26.8	12.7	24.1	43.2	19.1
64	14.6	26.1	11.5	20.8	36.0	15.2
128	20.5	32.7	12.2	24.4	38.3	13.9

Table 3.1 Time in seconds of the UPC NPB-CG benchmark with and without deadlock checking

Threads	Class B			Class C		
	Without checks	With checks	Overhead	Without checks	With checks	Overhead
16	0.76	1.03	0.27	2.72	3.87	1.15
32	0.78	0.93	0.15	2.65	3.17	0.52
64	0.86	0.94	0.08	2.20	2.52	0.32
128	0.96	1.04	0.08	1.94	2.13	0.19

Table 3.2 Time in seconds of the UPC NPB-IS benchmark with and without deadlock checking

technique to check deadlock errors in collective operations and uses a distributed wait-for-graph for detecting deadlocks involving locks. The algorithmm has been proven to be correct and to have a run-time complexity of O(1) when detecting errors in UPC collective operations. This algorithm has been extended to detect deadlocks involving locks with a run-time complexity of O(T), T is the number of threads involved in the deadlock. 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.

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.

CHAPTER 4. GENERAL CONCLUSIONS

Today's high performance programs could consists of thousands of lines of code, implement extremely complex algorithms, and run on hundreds of thousands of cores. Debugging such programs often requires intricate understanding of the program. This is often difficult with such large programs written by a team of people and over a period of time. Good quality error messages could enable programmers to fix the errors in their programs quickly.

UPC is an extension of C programming language for parallel execution on shared and distributed memory parallel machines using the Partitioned Global Address Space (PGAS) programming model. Previous work by Professor Glenn Luecke's High Performance Computing (HPC) Group at Iowa State University (ISU) found that most UPC compilers and run-time environments either fail to detect an error or provide very poor error messages.

As a remedy, a run-time error detection tool called UPC-CHECK has been developed. The tool can be used on any system where the user intends to build and execute UPC-programs. UPC-CHECK can handle argument errors and deadlock errors in UPC programs. It employs a novel optimal and distributed run-time algorithm to detect deadlocks created by collective operations in UPC. The algorithm has a run-time complexity of O(1) while detecting deadlocks created by errors in collective operations. The above algorithm is extended to detect deadlock conditions while trying to acquire locks by using a distributed shared Wait-For-Graph.

In addition to detecting argument errors and deadlock errors in UPC programs, UPC-CHECK can detect some errors involving wrong order of UPC operations, such as unmatched upc_notify and upc_wait statement and illegal control flow into or out of the body of a controlling upc_forall loop.

UPC-CHECK is very easy to use and comes with necessary documentation in the form of a User's Guide and Tutorial. A script is also provided to install the tool itself and all software that it is dependent on. The tool is primarily based on a source-to-source translator and is therefore machine and compiler independent. It has been extensively tested for hundreds of error tests as well as error-free tests.

With UPC-CHECK, a near complete coverage of possible argument errors and deadlock errors has been shown using comprehensive test suites. The error messages produced provide the line number, executing thread and file name where the error occurred as well as additional information that will help fix the detected error. Initial testing with real-life UPC programs like the UPC NAS parallel benchmark yielded favorable results regarding the efficiency and scalibility of UPC-CHECK. It is hoped that with the proven scalability and low overhead UPC-CHECK can provide a good programming environment to develop and debug UPC programs.

APPENDIX A. USER'S GUIDE FOR UPC-CHECK 1.0

Jim Coyle and Marina Kraeva.
High Performance Computing Group
Iowa State University

A.1 Background

UPC is an extension of C programming language for parallel execution on shared or distributed memory parallel machines[3] that uses the Partitioned Global Address Space (PGAS) programming model. UPC-CHECK 1.0 is a tool for the automatic detection of deadlocks and argument errors in UPC functions. UPC-CHECK was developed by Professor Glenn Luecke's High Performance Computing (HPC) Group at Iowa State University (ISU). Error messages issued by UPC-CHECK have been designed to help users quickly fix the problems detected. UPC-CHECK 1.0 was tested using the UPC Run-Time Error Detection test suite[13] developed by ISU'S HPC Group.

A.2 How to use UPC-CHECK

UPC-CHECK uses the ROSE Toolkit from Lawrence Livermore National Laboratory to instrument UPC source code. When the instrumented source code is run, run-time errors are detected and error messages are issued to help fix the errors. Error messages provide the line number, executing thread and file name where the error occurred as well as additional information that will help fix the detected error.

To instrument sourcefile.upc and compile the instrumented UPC program, issue

```
upc-check [compiler options] [--upccheck:flag [--upccheck:flag] ...]
```

```
    -c sourcefile.upc
```

where "compiler options" are options which will be passed to the UPC compiler after instrumentation. This will produce a file named `sourcefile.instrumented.o`.

To instrument sourcefile.upc, compile and link the instrumented UPC program issue

```
upc-check [compiler and upccheck:  options] -o a.out sourcefile.upc
```

This will produce an executable named a.out with the UPC-CHECK support routines automatically included. The executable created can then be run and the errors detected by UPC-CHECK will be reported. UPC-CHECK allows multiple source files, e.g.

```
upc-check [compiler and upccheck:  options] -o a.out sourcefile1.upc
    sourcefile2.upc
```

Notice that UPC-CHECK is used by merely replacing the compiler name with upc-check. UPC-CHECK automatically performs both deadlock and argument error checking, but each of these checks can be turned off to reduce running time for the instrumented executable if desired. The flags for the UPC-CHECK option, −−upccheck, are listed in the following table. Notice that there is a flag to enable UPC-CHECK to trace function call stacks.

`-a|-d_argument_check`	`disables argument checking (enabled by default)`
`-d|-d_deadlock_check`	`disables deadlock checking (enabled by default)`
`-s|-e_track_func_call_stack`	`enables tracing of function call stack (disabled by default)`
`-h|--h|-help`	`prints help for UPC-CHECK`

For example, to disable deadlock checking in UPC-CHECK, one would issue

```
upc-check -d_deadlock_check -o source.out sourcefile.upc
```

A.3 Environmental Variables:

UPCCHECK_STOP_ON_ERROR : By default, UPC-CHECK will stop on an error and continue on a warning. Setting the environmental variable UPCCHECK_STOP_ON_ERROR to FALSE will allow execution to continue when UPC-CHECK finds an error. This could be used to find more than one error in a single run.

UPCCHECK_LIVELOCK_TIMEOUT : In some cases, a process may loop on upc_lock_attempt forever, because another thread holds the lock. This is not a deadlock because at least one thread may continue executing. To contrast this from deadlock, this is called a livelock. To detect this case, we use a timeout value 300 seconds from the time a upc_lock_attempt is issued until it is satisfied. If the upc_lock_attempt is not satisfied within that time, an error message will be printed to STDERR reporting that a livelock condition may be present. This message will include the upc_lock_attempt location and the lock involved. The environmental variable UPCCHECK_LIVELOCK_TIMEOUT allows the timeout value to be changed from the default value of 300 to some other number of seconds.

A.4 Installation Guide for UPC-CHECK 1.0

To use UPC-CHECK, one must already have a UPC compiler installed. The install_UPC-CHECK script below assumes that a UPC compiler is installed and is in the path of the user who is performing the install. The procedure checks first for upcc, which is LLNL Berkley UPC, then for upc which is either GNU-UPC or HP UPC, and then defaults to cc which is used for the Cray UPC compiler.

UPC-CHECK uses the ROSE Toolkit [40] and the ROSE Toolkit uses BOOST [1]. If BOOST and/or ROSE are not already installed, the afore-mentioned websites contain download and installation information. UPC-CHECK is installed by issuing the following:

STEP 1: `wget http://hpcgroup.public.iastate.edu/UPC-CHECK/UPC-CHECK.tar.gz`

STEP 2: `tar -zxf UPC-CHECK.tar.gz`

STEP 3: `cd UPC-CHECK`

STEP 4: `./install_UPC-CHECK -p INSTALL_DIR -b BOOST_DIR -r ROSE_DIR`

If `-p INSTALL_DIR` is omitted, this is the same as `-p /usr/local` , similarly for `-b` and `-r`.

Once installed, the UPC-CHECK executable can be copied form `INSTALL_DIR/bin` to a system bin directory, like `/usr/local/bin` if desired.

A.5 Tutorial:

The following examples illustrate how to use UPC-CHECK to find and correct program errors. Errors issued by UPC-CHECK are independent of which UPC compiler used. (Berkeley's UPC compiler was used for these examples.)

Example 1: Error in the value passed to a UPC function.

```
cat -n ex1.upc
1     #include <upc.h>
2     #include <stdio.h>
3     #include <stdlib.h>
4     #include <upc_collective.h>
5
6     #define BLOCK_SIZE 3
7
8     shared [BLOCK_SIZE] int arrA[THREADS * BLOCK_SIZE];
9     shared [BLOCK_SIZE] int arrB[THREADS * BLOCK_SIZE];
10    static shared int sh_val;
11
12    int main() {
13       int num_bytes, i ;
14
15       /* The programmer forgot to include the following line */
16       /* sh_val=BLOCK_SIZE; */
17       upc_forall(i=0; i<THREADS*BLOCK_SIZE; i++; (i/BLOCK_SIZE)) {
18          arrA[i] = (i+MYTHREAD) * (i-MYTHREAD) + 1;
19       }
20       upc_forall(i=0; i<THREADS*BLOCK_SIZE; i++; (i/BLOCK_SIZE)) {
```

```
21          arrB[i] = (i+MYTHREAD) * (i-MYTHREAD) + 1;
22       }
23
24       upc_barrier;
25
26       upc_all_broadcast(arrA, arrB, sizeof(int)*sh_val, UPC_IN_NOSYNC |
                                         UPC_OUT_NOSYNC);
27
28       upc_barrier;
29
30       if (MYTHREAD == 0) {
31          for(i=0;i<THREADS*BLOCK_SIZE; i++) {
32             printf(‘‘Thread %d arrB[%d] / arrA[%d]=%d \n’’,
33             MYTHREAD, i,i,arrB[i]/arrA[i]);
34          }
35       }
36
37       return 0;
38    }
```

Using UPC-CHECK as follows

```
> upc-check -T 4 -o ex1 ex1.upc > time upcrun -n 4 ./ex1
```

produces the following message:

```
Thread 0 encountered invalid arguments in function upc_all_broadcast at line 26
in file /home/jjc/ex1.upc.
Error:  Parameter (((sizeof(int )) *(sh_val))) passes non-positive value of 0 to
nbytes argument
Variable sh_val was declared at line 10 in file /home/jjc/ex1.upc.
44.162u 0.024s 0:12.20 362.1%  0+0k 0+0io 0pf+0w
```

UPC-CHECK has detected that in line 26 of ex1.upc on thread 0, the value of the expression being passed: “sizeof(int)*sh_val” is zero. This is not allowed in the argument of upc_all_broadcast which has the formal parameter nbytes.

UPC-CHECK also reports that the variable sh_var was declared in line 10. Notice that it is declared as a static shared global variable. Since no initialization of this value has occurred before it is used in line 26, it would have the default (incorrect) value 0 as reported. When the statement, sh_val=BLOCK_SIZE, is inserted at line 16, the program works as intended.

Example 2: Failure to allocate hints passed to upc_all_fopen when numhints>0, causing file not to open.

```
> cat -n ex2.upc
1      #include <upc.h>
2      #include <stdio.h>
3      #include <stdlib.h>
4      #include <upc_io.h>
5
6         int main() {
7         size_t numhints;
8         upc_file_t *fd;
9         struct upc_hint *hints;
10
11        numhints = 1;
12
13        fd = upc_all_fopen(‘‘upcio1.txt’’,UPC_INDIVIDUAL_FP|UPC_WRONLY|UPC_CREATE,
                                numhints, hints);
14
15        if (fd != NULL)
16           upc_all_fclose(fd);
17        else
18          if(!MYTHREAD) printf(‘‘File not open \n’’);
19
20        return 0;
21    }
```

Using UPC-CHECK as follows

```
> upc-check -T 4 -o ex2 ex2.upc
> upcrun -n 4 ./ex2
```

produces the following message:

```
Thread 0 encountered invalid arguments in function upc_all_fopen at line 13
in file /home/jjc/ex2.upc.
Error:  Parameter numhints passes non-zero value of 1 to ‘numhints’ argument
while target of parameter (hints) passed to ‘hints’ argument is unallocated.
Variable numhints was declared at line 7 in file /home/jjc/ex2.upc.
Variable hints was declared at line 9 in file /home/jjc/ex2.upc.
```

In line 13, we see that one hint was to be passed to the I/O routine, but hints was not pointing to allocated space. (*hints should have been allocated and initialized.) Therefore, either numhints must be set to 0 or *hints must be allocated and initialized.

Example 3: Failure to call upc_barrier from all threads.

```
> cat -n ex3.upc 1      #include <upc.h>
2      #include <math.h>
3      #include <stdio.h>
4      #include <stdlib.h>
5
6      /* function that returns an integer zero value is unlikely to be computed
       at compile-time */
7      int zero(){
8         return (int) (sin(0.1*MYTHREAD)/2.3);
9      }
```

```

extern void funcA();

int i, sum;
shared int arrA[THREADS];

int main() {

   arrA[MYTHREAD] = MYTHREAD;

   /* called by thread 1 only */
   if (MYTHREAD == 1) {
      /* this should never be executed */
      if (zero()) {
         funcA();
      }
   }

   /* called by all other threads */
   else {
      funcA();
   }

   /* sum up all elements in arrA */
   sum = 0; for (i = 0; i < THREADS; i++) { sum += arrA[i]; }
   if ((MYTHREAD==0) || zero()) printf(‘‘thread %i:  value = %i\n’’,
                                       MYTHREAD, (int) sum);
   /* end reached - terminate */
38       if ((MYTHREAD==0) || zero()) printf(‘‘thread %i:  end\n’’, MYTHREAD);
39       return 0;
40    }
```

```
> cat -n /home/jjc/ex3_s.upc
1      #include <upc.h>
2      #include <math.h>
3      #include <stdio.h>
4      #include <stdlib.h>
5
6      void funcA() {
7         upc_barrier;
8      }
```

Using UPC-CHECK as follows `> upc-check -T 4 -o ex3 ex3.upc ex3_s.upc`

```
> upcrun -n 4 ./ex3
```

produces the following message:

```
Runtime error:  Deadlock condition detected:  One or more threads have finished
executing while other threads are waiting at a collective routine
Status of threads
=================
Thread id:Status:Presently waiting at line number:of file
---------------------------------------------------------
0:waiting at upc_barrier:  7:  /home/jjc/ex3_s.upc
1:reached end of execution through:  39:  /home/jjc/ex3.upc
2:executing:unknown:unknown
3:executing:unknown:unknown
```

We see that the call to upc_barrier at line 7 in ex3_s.upc is missing for thread 1. Since every thread that enters funcA() calls upc_barrier, thread 1 must not have called funcA(). Looking at where thread 1 calls funcA() in ex3.upc, we can see that funcA() may or may not be called depending on whether the body of the following if statement is executed.

```
23          if (zero()) {
24             funcA();
25          }
```

Clearly it must not be for this run, so one can look either at why the function zero() is returning 0 on thread 1 or whether upc_barrier should be called in an else-block for this if.

Example 4: Deadlock due to missing collective routine.

```
> cat -n ex4.upc
1      #include <upc.h>
2      #include <math.h>
3      #include <stdio.h>
4      #include <stdlib.h>
5
6      /* function that returns an integer zero value which can not be calculated
at compile-time */
7      int zero(){
8         return (int) (sin(0.1*MYTHREAD)/2.3);
9      }
10
11     #include <upc_io.h>
12
13     #define BLOCKSIZE 4
14
15     shared [BLOCKSIZE] char tempArray[5 * THREADS];
16
17     extern void syncFiles(upc_file_t * fd);
18
19     int main()
```

```
{
   int i;
   upc_file_t *fd;

   upc_forall(i = 0; i < 20; i++; &tempArray[i])
   {
      tempArray[i] = 'a'+ i;
   }

   upc_barrier;

   fd = upc_all_fopen( ''c_J_1_1_n.txt'',
               UPC_WRONLY|UPC_COMMON_FP|UPC_CREATE,
               0, NULL );

   upc_all_fwrite_shared(fd, tempArray,
               BLOCKSIZE, sizeof(char), 20,
               UPC_IN_ALLSYNC | UPC_OUT_NOSYNC );

   /* called by thread 1 only */
   if (MYTHREAD == 1) {
      /* this should never be executed */
      if (zero()) {
         syncFiles(fd);
      }
   }

   /* called by all other threads */
48        else {
49           syncFiles(fd);
50        }
51
52        upc_all_fclose(fd);
53
54        return 0;
55     }
> cat -n ex4_s.upc
1      #include <upc.h>
2      #include <math.h>
3      #include <stdio.h>
4      #include <stdlib.h>
5      #include <upc_io.h>
6
7      void syncFiles(upc_file_t * fd)
8      {
9         upc_all_fsync(fd);
10     }
```

Using UPC-CHECK as follows

```
> upc-check -T 4 -o ex4 ex4.upc ex4_s.upc
> time upcrun -n 4 ./ex4
```

produces the following message:

```
Runtime error:  Deadlock condition detected:  Different threads waiting at
different collective routines
Status of threads
=================
Thread id:Status:Presently waiting at line number:of file
```

```
---------------------------------------------------------
0:waiting at upc_all_fsync on file pointer fd:  9:  /home/jjc/ex4_s.upc
1:waiting at upc_all_fclose on file pointer fd:  52:  /home/jjc/ex4.upc
2:executing:unknown:unknown
3:executing:unknown:unknown
43.450u 0.020s 0:12.21 356.0%  0+0k 0+0io 0pf+0w
```

This message reports that the program is deadlocked and that two different threads are waiting at different collective routines. Thread 0 is stopped at a call to upc_all_fsync at line 9 of ex4_s.upc. Looking at the file ex4_s.upc, one can see that this line is in the function syncFiles. syncFiles is called at two different lines in ex4.upc, both of which will always precede the point of execution where thread 1 is reported to be stopped (line 52 of ex4.upc). Thus, thread 0 called syncFiles but thread 1 did not. Looking at the if structure in lines 40-50 of ex4.upc, we see:

```
40        if (MYTHREAD == 1) {
41           /* this should never be executed */
42           if (zero()) {
43              syncFiles(fd);
44           }
45        }
46
47        /* called by all other threads */
48        else {
49           syncFiles(fd);
50        }
```

so for thread 0, syncFiles is called only if the body of the if block executes.

```
42           if (zero()) {
43              syncFiles(fd);
44           }
```

Clearly, the program logic must be changed. Perhaps upc_all_fsync(fd) should be called in an else-block of this if.

Example 5: Different source arrays are passed to upc_all_reduce function.

```
> cat -n ex5.upc
1      #include <stdlib.h>
2      #include <stdio.h>
3      #include <upc.h>
4      #include <upc_collective.h>
5
6      #define N 4
7      shared [N] int *ptrA;
8      shared int sumA;
9
10     int main() {
11     int i;
12
13        ptrA = upc_global_alloc(THREADS,N*sizeof(int));
14
15        upc_forall(i=0; i<N*THREADS;i++;i/N) {
16           ptrA[i] = i;
17        }
18
19        upc_barrier;
20
21        upc_all_reduceI(&sumA, ptrA, UPC_ADD, N*THREADS, N, NULL,
             UPC_IN_NOSYNC|UPC_OUT_NOSYNC);
22
23        upc_barrier;
24
25       if (MYTHREAD==0) printf(‘‘sumA=%d\n’’,sumA);
26
27       return 0;
28    }
```

Using UPC-CHECK as follows

```
> upc-check -T 4 -o ex5 ex5.upc
> time upcrun -n 4 ./ex5
```

produces the following message:

```
Runtime error:  Unspecified behavior condition detected, may lead to deadlock :
One or more threads have different values for single_valued parameters.
Status of threads
=================
Thread id:Status:Presently waiting at line number:of file
---------------------------------------------------------
0:waiting at upc_all_reduceI: 21:  /home/jjc/ex5.upc
1:waiting at upc_all_reduceI: 21:  /home/jjc/ex5.upc
2:executing:unknown:unknown
3:waiting at upc_all_reduceI: 21:  /home/jjc/ex5.upc
Mismatch in parameter:  src.
Thread no.
=====================================================================
0:(ptrA) points to memory location 0x2acf62c46ff0.
Variable ptrA was declared at line 7 in file /home/jjc/ex5.upc.
1:(ptrA) points to memory location 0x2acf62c46fc0.
Variable ptrA was declared at line 7 in file /home/jjc/ex5.upc.
2:
3:(ptrA) points to memory location 0x2acf62c46fe0.
```

```
Variable ptrA was declared at line 7 in file /home/jjc/ex5.upc.
44.402u 0.068s 0:12.28 362.0% 0+0k 0+0io 0pf+0w
```

This message reports that threads have different values of the src parameter of function upc_all_reduceI. ptrA, declared at line 7 of file ex5.upc, points to different memory locations. Looking at the ptrA declaration, we see that ptrA is a private pointer-to-shared. Later in the code its assigned the value returned by the call to upc_global_alloc. This function is not collective. If its called by multiple threads, all threads which make the call get different allocations. The author of the code probably meant to call upc_all_alloc instead. Note that with the current version of Berkley UPC compiler, the value of sumA will be the same in either case, but this behavior is not guaranteed for the test above.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincerest gratitude to those who helped me with various aspects of conducting this research and writing this thesis.

First and foremost, I would like to thank Dr. Glenn R. Luecke for his guidance, patience and support throughout the expanse of this research and the writing of this thesis. His insights and words of encouragement have often inspired and reinvigorated me.

I would also like to thank my committee members Dr. Suraj C. Kothari and Dr. Zhao Zhang for their valuable time, suggestions and contributions towards this body of work.

I would always be indebted to Dr. Marina Kraeva and Dr. James Coyle for their guidance and inputs throughout the entire duration of the project. Dr. Marina and Dr. James not only helped in the design of the project, but also provided valuable inputs, alternatives and modifications throughout the implementation and testing of the project.

I am also thankful to Dr. Jim Hoekstra for providing me with the clusters along with the required software for the implementation of this project.

I would like to express my gratitude to Dr. Dan Quinlan, and his colleagues working at the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory for their ROSE compiler infrastructure. They not only cleared my queries about ROSE in a timely fashion, but also promptly provided enhancements and bug-fixes required for this project.

BIBLIOGRAPHY

[1] BOOST.

[2] High Performance Computing (HPC) Group, Iowa State University.

[3] The High Performance Computing Laboratory, The George Washington University.

[4] Sun Microsystems HPC ClusterTools.

[5] The Berkeley Unified Parallel C.

[6] Unified Parallel C (Wikipedia).

[7] UPC NAS Parallel Benchmarks.

[8] Gabriel Bracha and Sam Toueg. Distributed deadlock detection. *Distributed Computing*, 2:127–138, 1987. 10.1007/BF01782773.

[9] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1:144–156, May 1983.

[10] Sébastien Chauvin, Proshanta Saha, François Cantonnet, Smita Annareddy, and Tarek El-Ghazawi. UPC manual. 2005.

[11] Shigang Chen, Yi Deng, P. Attie, and Wei Sun. Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 613 –619, may 1996.

[12] A.N. Choudhary, W.H. Kohler, J.A. Stankovic, and D. Towsley. A modified priority based probe algorithm for distributed deadlock detection and resolution. *Software Engineering, IEEE Transactions on*, 15(1):10 –17, jan 1989.

[13] James Coyle, James Hoekstra, Marina Kraeva, Glenn R. Luecke, Elizabeth Kleiman, Varun Srinivas, Alok Tripathi, Olga Weiss, Andre Wehe, Ying Xu, and Melissa Yahya. UPC run-time error detection test suite. 2008.

[14] James Coyle, Indranil Roy, Marina Kraeva, and Glenn R. Luecke. UPC-CHECK: A scalable tool for detecting run-time errors in Unified Parallel C. In *Proceedings of International Supercomputing Conference (ICS)*, June 2012. to appear.

[15] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel®message checker. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM.

[16] Ali Ebnenasir. UPC-SPIN: A Framework for the Model Checking of UPC Programs. In *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS '11, 2011.

[17] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming.* Wiley-Interscience, 2003.

[18] V.D. Gligor and S.H. Shattuck. On deadlock detection in distributed systems. *Software Engineering, IEEE Transactions on*, SE-6(5):435 – 440, sept. 1980.

[19] Iowa State University High Performance Computing Group. User's Guide for UPC-CHECK 1.0. 2011.

[20] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. A graph based approach for MPI deadlock detection. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 296–305, New York, NY, USA, 2009. ACM.

[21] Tobias Hilbrich, Martin Schulz, Bronis R. Supinski, and Matthias S. Müller. Must: A scalable approach to runtime error detection in mpi programs. In Matthias S. Müller,

Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 53–66. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-11261-4_5.

[22] S. T. Huang. A distributed deadlock detection algorithm for csp-like communication. *ACM Trans. Program. Lang. Syst.*, 12:102–122, January 1990.

[23] Joo Kyun Kim and Kern Koh. A 0(1) time deadlock detection scheme in a single unit and single request multiprocessor system. In *TENCON '91.1991 IEEE Region 10 International Conference on EC3-Energy, Computer, Communication and Control Systems*, volume 2, pages 219 –223, aug 1991.

[24] Edgar Knapp. Deadlock detection in distributed databases. *ACM Comput. Surv.*, 19:303–328, December 1987.

[25] Kraeva, Marina and Coyle, James and Luecke, Glenn R. and Roy, Indranil and Kleiman, Elizabeth and Hoekstra, Jim. UPC-CompilerCheck: A Tool for Evaluating Error Detection Capabilities of UPC Compilers. preprint (2012).

[26] Bettina Krammer, Matthias Müller, and Michael Resch. MPI application development using the analysis tool marmot. In Marian Bubak, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 464–471. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24688-6_61.

[27] A.D. Kshemkalyani and M. Singhal. Efficient detection and resolution of generalized distributed deadlocks. *Software Engineering, IEEE Transactions on*, 20(1):43 –54, jan 1994.

[28] Soojung Lee. Fast, centralized detection and resolution of distributed deadlocks in the generalized model. *Software Engineering, IEEE Transactions on*, 30(9):561 – 573, sept. 2004.

[29] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, and Indranil Roy. UPC-CHECK Tutorial. 2011.

[30] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Elizabeth Kleiman, and Olga Weiss. Evaluating error detection capabilities of UPC run-time systems. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 7:1–7:4, New York, NY, USA, 2009. ACM.

[31] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Mi-Young Park, Elizabeth Kleiman, Olga Weiss, Andre Wehe, and Melissa Yahya. The importance of run-time error detection. In Matthias S. Muller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 145–155. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-11261-4_10.

[32] Glenn R. Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[33] Jayadev Misra and K. M. Chandy. A distributed graph algorithm: Knot detection. *ACM Trans. Program. Lang. Syst.*, 4:678–686, October 1982.

[34] Don P. Mitchell and Michael J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 282–284, New York, NY, USA, 1984. ACM.

[35] J. Eliot B. Moss. Nested transactions: An approach to reliable distributed computing, 1981.

[36] N Natarajan. A distributed scheme for detecting communication deadlocks. *IEEE Trans. Softw. Eng.*, 12:531–537, April 1986.

[37] Ron Obermarck. Distributed deadlock detection algorithm. *ACM Trans. Database Syst.*, 7:187–208, June 1982.

[38] Paul Petersen and Sanjiv Shah. OpenMP support in the Intel®Thread Checker. In Michael Voss, editor, *OpenMP Shared Memory Parallel Programming*, volume 2716 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-45009-2_1.

[39] Peter Pirkelbauer, Chunhua Liao, Thomas Panas, and Daniel Quinlan. Runtime detection of c-style errors in upc code. In *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS '11, 2011.

[40] Daniel J. Quinlan and et al. ROSE compiler project.

[41] M. Roesler and W.A. Burkhard. Resolution of deadlocks in object-oriented distributed systems. *Computers, IEEE Transactions on*, 38(8):1212 –1224, aug 1989.

[42] Indranil Roy. UPC-CHECK: A scalable tool for detecting run-time errors in Unified Parallel C. Master's thesis, Iowa State University, Ames, Iowa, USA, 2012. Preprint.

[43] Indranil Roy, Glenn R. Luecke, James Coyle, Marina Kraeva, and James Hoekstra. An optimal deadlock detection algorithm for Unified Parallel C. preprint (2012).

[44] P.H. Shiu, Yudong Tan, and III Mooney, V.J. A novel parallel deadlock detection algorithm and architecture. In *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, pages 73 –78, 2001.

[45] Stephen Siegel. Verifying parallel programs with mpi-spin. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-75416-9_8.

[46] M.K. Sinha and N. Natarajan. A priority based distributed deadlock detection algorithm. *Software Engineering, IEEE Transactions on*, SE-11(1):67 – 80, jan. 1985.

[47] The UPC Consortium. UPC Language Specifications (v1.2). 2005.

[48] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert 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*, CAV '08, pages 66–79, Berlin, Heidelberg, 2008. Springer-Verlag.

[49] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert 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*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM.

[50] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[51] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B.R. 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*, pages 1 –10, nov. 2010.

[52] Xiang Xiao and J.J. Lee. A novel parallel deadlock detection algorithm and hardware for multiprocessor system-on-a-chip. *Computer Architecture Letters*, 6(2):41 –44, feb. 2007.

[53] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[54] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the*

2007 international workshop on Parallel symbolic computation, PASCO '07, pages 24–32, New York, NY, USA, 2007. ACM.