myjournal manuscript No.
(will be inserted by the editor)

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

James Coyle · Indranil Roy · Marina Kraeva · Glenn R. Luecke

Received: date / Accepted: date

Abstract Unified Parallel C (UPC) [23] is a language used to write parallel programs for 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 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 [6] 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.

Keywords UPC · run-time error detection · distributed deadlock detection · partitioned global address space

1 Introduction

The importance of error detection is well documented by Glenn Luecke et al. [16]: "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) [2, 5, 9] is poor [14, 15].

Though tools to detect errors in serial, MPI [7, 11, 12, 17, 24] and OpenMP programs [3, 18] 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 [19] and UPC-SPIN [8]. 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 [8] 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) [4] 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 3.2 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.

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

Iowa State University's High Performance Computing Group, Iowa State University, Ames, Iowa 50011, USA.
E-mail: jjc@iastate.edu · iroy@iastate.edu · kraeva@iastate.edu · grl@iastate.edu

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 [21, 22].

Section 2 provides an overview of UPC-CHECK including its design, functionality and usage. Section 3 describes the function, scalability, overhead and compiler-independence testing of UPC-CHECK. Section 5 contains an example illustrating how UPC-CHECK can be used to find and correct a deadlock in a UPC program.

2 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 [20] 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 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.

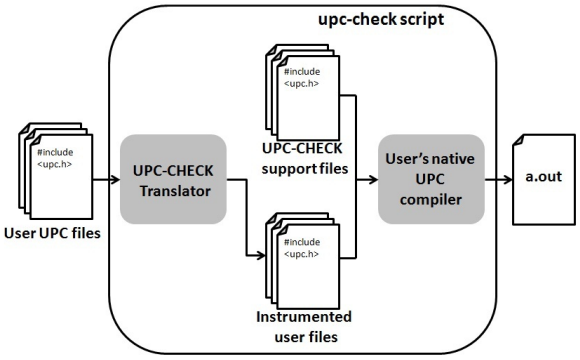

Fig. 1 Compiling files with UPC-CHECK

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 [13] and user's guide [10].

2.1 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 Section2.2.

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.

2.2 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.

2.2.1 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.

2.2.2 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 [21, 22], 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. 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.

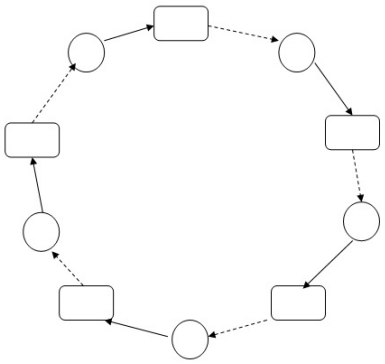

Fig. 2 Circular dependencies of threads leading to a deadlock.

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.

Thirdly, deadlocks could also be created when there is a chain of hold-and-wait dependencies that can be

Fig. 3 Chain of dependencies leading to a thread that is either waiting at a collective operation or has completed execution.

resolved by unlocking a lock which is held by a thread blocked at a collective operation, see Figure 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.

2.2.3 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.

3 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.

3.1 Function testing

UPC-CHECK was tested against over 350 test cases in the UPC RTED test suite [6]. 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 [15], the High Performance Computing (HPC) Group at Iowa State University (ISU) [1] 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 1.

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

Table 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 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 2.2.3.

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 [21, 22].

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.

3.2 Scalability and overhead testing

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

3.2.2 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.

4 Compiler-independence testing

UPC-CHECK was tested for compiler-independence using all function tests in Section 3.1 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 [10].

5 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 wait-
ing 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.

6 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 [6] 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.

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

```
...
6 void funcA() {
7       upc_barrier;
8 }
...
```

References

1. High Performance Computing (HPC) Group, Iowa State University. URL `http://www.it.iastate.edu/research/hpcg/`
2. The High Performance Computing Laboratory, The George Washington University. URL `http://upc.gwu.edu`
3. Sun Microsystems HPC ClusterTools. URL `http://www.sun.com/software/products/clustertools`
4. UPC NAS Parallel Benchmarks. URL `http://threads.hpcl.gwu.edu/sites/npb-upc`

5. Chauvin, S., Saha, P., Cantonnet, F., Annareddy, S., El-Ghazawi, T.: UPC manual (2005). URL http://upc.gwu.edu/downloads/Manual-1.2.pdf
6. Coyle, J., Hoekstra, J., Kraeva, M., Luecke, G.R., Kleiman, E., Srinivas, V., Tripathi, A., Weiss, O., Wehe, A., Xu, Y., Yahya, M.: UPC run-time error detection test suite. URL http://kraeva.public.iastate.edu/rted/UPC.TestPlan.pdf
7. DeSouza, J., Kuhn, B., de Supinski, B.R., Samofalov, V., Zheltov, S., Bratanov, S.: 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, pp. 78–82. ACM, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1145319.1145342. URL http://doi.acm.org/10.1145/1145319.1145342
8. Ebnenasir, A.: 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). URL http://pgas11.rice.edu/papers/PirkelbauerEtAl-UPC-Error-Detect-PGAS11.pdf
9. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.: UPC: Distributed Shared Memory Programming. Wiley-Interscience (2003)
10. High Performance Computing Group, I.S.U.: User's Guide for UPC-CHECK 1.0 (2011). URL http://hpcgroup.public.iastate.edu/UPC-CHECK/UPC-CHECK_UsersGuide.pdf
11. Hilbrich, T., de Supinski, B.R., Schulz, M., Müller, M.S.: A graph based approach for MPI deadlock detection. In: Proceedings of the 23rd international conference on Supercomputing, ICS '09, pp. 296–305. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1542275.1542319. URL http://doi.acm.org/10.1145/1542275.1542319
12. Krammer, B., Müller, M., Resch, M.: MPI application development using the analysis tool marmot. In: M. Bubak, G. van Albada, P. Sloot, J. Dongarra (eds.) Computational Science - ICCS 2004, *Lecture Notes in Computer Science*, vol. 3038, pp. 464–471. Springer Berlin / Heidelberg (2004). URL http://dx.doi.org/10.1007/978-3-540-24688-6_61. 10.1007/978-3-540-24688-6_61
13. Luecke, G., Coyle, J., Hoekstra, J., Kraeva, M., Roy, I.: UPC-CHECK Tutorial (2011). URL http://hpcgroup.public.iastate.edu/UPC-CHECK/UPC-CHECK_Tutorial_Aug_30.pptx
14. Luecke, G.R., Coyle, J., Hoekstra, J., Kraeva, M., Kleiman, E., Roy, I.: Evaluating error detection capabilities of UPC compilers. URL http://hpcgroup.public.iastate.edu/papers/UPC.CTED.Paper.pdf. Preprint (2010)
15. Luecke, G.R., Coyle, J., Hoekstra, J., Kraeva, M., Xu, Y., Kleiman, E., Weiss, O.: Evaluating error detection capabilities of UPC run-time systems. In: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, PGAS '09, pp. 7:1–7:4. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1809961.1809971. URL http://doi.acm.org/10.1145/1809961.1809971
16. Luecke, G.R., Coyle, J., Hoekstra, J., Kraeva, M., Xu, Y., Park, M.Y., Kleiman, E., Weiss, O., Wehe, A., Yahya, M.: The importance of run-time error detection. In: M.S. Muller, M.M. Resch, A. Schulz, W.E. Nagel (eds.) Tools for High Performance Computing 2009, pp. 145–155. Springer Berlin Heidelberg (2010). URL http://dx.doi.org/10.1007/978-3-642-11261-4_10. 10.1007/978-3-642-11261-4_10
17. Luecke, G.R., Zou, Y., Coyle, J., Hoekstra, J., Kraeva, M.: Deadlock detection in MPI programs. Concurrency and Computation: Practice and Experience **14**(11), 911–932 (2002). DOI 10.1002/cpe.701. URL http://dx.doi.org/10.1002/cpe.701
18. Petersen, P., Shah, S.: OpenMP support in the Intel®Thread Checker. In: M. Voss (ed.) OpenMP Shared Memory Parallel Programming, *Lecture Notes in Computer Science*, vol. 2716, pp. 1–12. Springer Berlin / Heidelberg (2003). URL http://dx.doi.org/10.1007/3-540-45009-2_1. 10.1007/3-540-45009-2_1
19. Pirkelbauer, P., Liao, C., Panas, T., Quinlan, D.: Runtime detection of c-style errors in upc code. In: Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models, PGAS '11 (2011). URL http://pgas11.rice.edu/papers/PirkelbauerEtAl-UPC-Error-Detect-PGAS11.pdf
20. Quinlan, D.J., et al.: ROSE compiler project. URL http://www.rosecompiler.org/
21. Roy, I.: 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
22. Roy, I., Luecke, G.R., Coyle, J., Kraeva, M., Hoekstra, J.: An optimal deadlock detection algorithm for Unified Parallel C. URL http://hpcgroup.public.iastate.edu/papers/Deadlock_Dectection_for_UPC.pdf. Preprint (2012)
23. The UPC Consortium: UPC Language Specifications (v1.2) (2005). URL http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf
24. Vetter, J.S., de Supinski, B.R.: Dynamic software testing of MPI applications with umpire. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '00. IEEE Computer Society, Washington, DC, USA (2000). URL http://dl.acm.org/citation.cfm?id=370049.370462