D^2C: Deterministic, Deadlock-free Concurrency

Nalini Vasudevan *, Stephen A. Edwards
Columbia University, New York
naliniv, sedwards@cs.columbia.edu

Julian Dolby, Vijay Saraswat
IBM Research, Hawthorne
dolby, vsaraswa@us.ibm.com

Abstract

The advent of multicore processors has made concurrent programming languages mandatory. However, most concurrent programming models come with two major pitfalls: non-determinism and deadlocks. By determinism, we mean the output behavior of the program is independent of the scheduling choices (e.g., the operating system) and depends only on the input behavior. A few concurrent programming models provide deterministic behavior by providing constructs that impose additional synchronization, but the improper (or the out of order) use of these constructs leads to problems like deadlocks.

In this paper, we argue for both determinism and deadlock-freedom and provide a deterministic, deadlock-free concurrent model. The model can be implemented either as programming language constructs or as a library. Any program that uses this model is guaranteed to produce the same output for a given input. Additionally, the program will never deadlock: the program will either terminate or run for ever.

1. Introduction

Non-deterministic functional behavior arising from timing variability is one of the biggest problems of concurrent programming. The program in Figure 1 is non-deterministic. It uses Cilk[1]-like syntax. It creates two tasks f and g in parallel using the *spawn* construct. Clearly, x is getting modified concurrently by both the tasks, so the value printed by this program is either 3 or 5 depending on the schedule.

Such non-determinism makes debugging all but impossible because unwanted behavior is rarely reproducible. Re-running a non-deterministic program on the same input usually does not produce the same behavior. We agree with Bocchino et al. [8] that the programming environment should ensure input-output determinism.

```
1  void f(shared int a) {
2    a = 3;
3  }
4
5  void g(shared int b) {
6    b = 5;
7  }
8
9  main() {
10   shared int x = 1;
11   spawn f(x)
12   spawn g(x);
13   sync; /* Wait for f and g to finish */
14   print x;
15 }
```

Figure 1. A non-deterministic parallel program

A few languages provide determinism through their semantics. They provide determinism by providing additional synchronization constructs. SHIM [6] for example provides determinism by special constructs, but these constructs can be used in such a way that a program in SHIM may deadlock. StreamIt [12] is another programming language that is explicitly deterministic but is suitable only for streaming applications and is a strict subset of SHIM.

In this paper, we propose a more flexible deterministic model, that can be either implemented as a language, programming language constructs or a library. The model is also deadlock-free: the synchronization is in such a way that any program that uses this model will never deadlock.

2. Approach

Non-determinism arises primarily due to read-write and write-write conflicts. In the D^2C model, we allow multiple tasks to write to a shared variable con-

* Student Author

```
1  void f(shared int &a) {
2   /* a is 1 */
3     a = 3;
4   /* a is 3 , x is still 1 */
5     next; /* The reduction operator is applied */
6   /* a is now 8, x is 8 */
7  }
8
9  void g(shared int &b) {
10  /* b is 1 */
11    b = 5;
12  /* b is 5, x is still 1 */
13    next; /* The reduction operator is applied */
14  /* b is now 8, x is 8 */
15 }
16
17 void h (shared int &c) {
18  /* c is 1 , x is still 1 */
19    next;
20  /* c is now 8, x is 8 */
21 }
22
23 main() {
24    shared int (+) x = 1;
25    /* If there are multiple writers, reduce
26       using the + reduction operator */
27    spawn f(x);
28    spawn g(x);
29    spawn h(x);
30    sync;
31    /* x is 8 */
32 }
```

Figure 2. A D^2C program

currently, but we define a commutative, associative reduction operator that will operate on these writes.

The program in Figure 2 creates three tasks in parallel f, g and h. f and g are modifying x. For simplicity, we have used Cilk[1]-like syntax. Even though f and g are modifying x concurrently, f sees the effect of g only when it executes *next*. Similarly g sees the effect of f only when it executes *next*. When a task executes *next*, it waits for all tasks that share variables with it, to also execute *next*. The *next* statement is like a barrier. At this statement, the shared variables are reduced using the reduction operator. In the example in Figure 2, the reduction operator is $+$ because x is declared with a reduction operator $+$ in line 24. Therefore after the *next* statement, the value of x is 3 + 5 which is 8 and it is reflected everywhere. Function h also rendezvous with f and g by executing *next* and thus it obtains the new value 8.

It is also possible to *not* define a reduction operator on a shared variable. Then, the first task among the spawned process (in program source order) overwrites the value. For example, in Figure 2, if the declaration of x in line 24 is *int x* rather than *int (+) x*, then x's value after *next* will be the value written by $f(x)$ which is 3. This is because $f(x)$ is the first concurrent process that is spawned.

The *next* synchronization statement is deadlock free. We do not give a formal proof here due to lack of space, but it follows from the fact that the *next* statement is a conjunctive barrier on all shared variables. On contrast, other deterministic concurrent models like SHIM are not deadlock free. Also, they do not allow multiple tasks to write to a shared variable because they provide ownership to variables.

3. Implementation

We implemented our model in the X10 programming language [4]. X10 is a parallel, distributed object-oriented language. To a Java-like sequential core it adds constructs for concurrency and distribution through the concepts of activities and places. An activity is a unit of work, like a thread in Java; a place is a logical entity that contains both activities and data objects. X10 uses the Cilk model of task parallelism and a task scheduler similar to that of Cilk.

Our preliminary implementation is as follows. We did a very conservative analysis to check if a particular shared variable is being used by multiple tasks. If yes, we force the variable to be shared with a reduction operator. This forces race-freedom. Otherwise, the compiler throws an error.

Each thread maintains a copy of the shared variable. A thread always reads from or writes to its local copy. Whenever the *next* statement is called, all threads sharing the variable synchronize. The last thread to synchronize does a linear reduction of the local copies using the commutative, associative operator in the variable declaration. It then updates the local copies with the new value.

4. Results

We ran a number of examples on a 2.33 GHz Intel Core 2 Duo with 2GB memory. Figure 3 shows the results. We measured the deterministic implementation of the applications with the original implementation. A

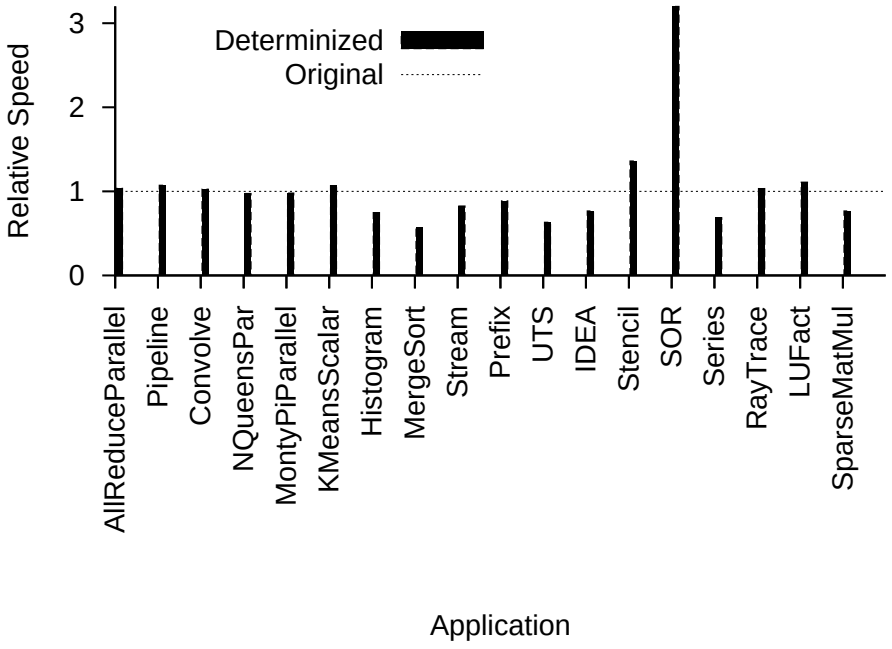

Figure 3. Experimental Results

bar with value below 1 indicates that the deterministic version ran slower than the original version.

The AllReduce Example is a parallel tree based implementation of reduction. The Pipeline example passes data through a number of intermediate stages; at each stage the data is processed and passed on to the next stage. Convolve is an application of the Pipeline program.

The N-Queens Problem finds the number of ways in which N queens can be placed on an N*N chessboard such that none of them attack each other. The MontiPi application finds the value of π using MonteCarlo simulation. The K-Means program partitions n data points into k clusters concurrently.

The Histogram program sorts an array into buckets based on the elements of the array. The Merge Sort program sorts an array of integers. The Prefix example operates on an array and the resulting array is obtained from the sum of the elements in the original array up to its index.

The SOR, IDEA, RayTrace, LUFact, SparseMatMul and Series programs are JGF benchmarks. The Raytracer benchmarks renders an image of sixty spheres. It has data dependent array access.

The SOR example performs Jacobi successive relaxation on a grid; it continuously updates a location of the grid based on the location's neighbors. The Stencil program is the 1-D version of the SOR.

The LUFact application transforms an N*N matrix into upper triangular form. The Series benchmark computes the first N coefficients of the function $f(x) = (x+1)^x$. The IDEA benchmark performs International Data Encryption algorithm (IDEA) encryption and decryption on an array of bytes. The SparseMatMul program performs multiplication of two sparse matrices.

The UTS benchmark [9] performing an exhaustive search on an unbalanced tree. It counts the number of nodes in the implicitly constructed tree that is parameterized in shape, depth, size, and imbalance.

For most of the examples, the deterministic version had a performance degradation of 1% - 25% as expected. However, for some examples like SOR and Stencil, the deterministic version performed better. The original version of these examples had explicit 2-phased barriers to differentiate between reads and writes, while the deterministic version requires just a single phase, because we maintain a local copy in each thread to eliminate read-write conflicts. Hence, the deterministic version performed better.

5. Related Work

There are a number of tools that provide determinism. For example, in the absence of data races, Kendo [10] ensures a deterministic order of all lock acquisitions for a given program input. However, if we have the sequence *lock(A); lock (B);* by one thread and *lock(B); lock(A);* by another thread, the deterministic ordering of locks could still lead to a deadlock. DMP [5] uses a deterministic token that is passed around all threads. A thread to modify a shared variable must first wait for the token and for all threads to block on that token. There is a lot of runtime overhead. Our method, although not discussed here, does a significant part of the work at compile time. Burmin and Sen [3] provide a framework for checking determinism for multithreaded programs. Their tool does not guarantee determinism because it is merely a testing tool that checks the execution trace with previously executed traces to see if the values match.

A few programming models provide explicit determinism. We have already discussed StreamIt [12] and SHIM [6]. Synchronous programming languages like Esterel are completely deterministic but they are highly restricted and are best suited for hardware systems.

Finally, type and effect systems like DPJ [2] have been designed for deterministic parallel programming. However, in general, type systems require the programmer to manually annotate the program. Our design does not require explicit annotation but provides restrictions through its constructs. One may argue for the need to learn a new programming paradigm or language, but we have done some work [17] to show that a model like this can be implemented as a library.

6. Conclusions and Future Work

We have presented a deterministic, deadlock free model. We have a proof (not shown here) that formulates this hypothesis. We have added these features as constructs to the X10 programming language. We also plan implement it as a library. A number of examples fit into this model: Histogram, Convolution, UTS, Sparse Matrix Multiplication etc.

Prior to this work, we designed compilers [7, 14] that generate deterministic code for shared-memory multicores and heterogeneous machines. We have also implemented a deterministic concurrent library [13] for the Haskell language. But these pieces of work were on SHIM, a model that is not guaranteed to be deadlock-free. So, we designed run-time [16] and static [13, 11] checkers to detect deadlocks at compile time. The D^2C model is a variant that provides deadlock-free determinism, and no special checkers are needed to detect deadlocks.

As future work, we plan to allow user defined reduction operators in our language. We therefore require a mechanism to check for associativity and commutativity of these operators. Secondly, we would like to use static analysis to improve the run-time efficiency of these constructs. Thirdly, we would like to implement this as a library, and check the program to see if it does not override the deterministic library. Next, we would like to build a determinizing tool [15] like Kendo [10] and [5] based on D^2C.

Our ultimate goal is efficient concurrency with determinism and deadlock-freedom. D^2C will introduce a way of bug-free parallel programming that will enable programmers to shift easily from sequential to parallel worlds and this will be a necessary step along the way to pervasive parallelism in programming.

Acknowledgement

I thank Stephen Edwards (Columbia University), Vijay Saraswat (IBM Research) and Julian Dolby (IBM Research) for contributing to parts of the work. The research was mainly supported by IBM Research and partly by NSF (grant 0614799).

References

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[2] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, New York, NY, USA, 2009. ACM.

[3] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of soft-*

ware engineering on European software engineering conference and foundations of software engineering symposium, pages 3–12, New York, NY, USA, 2009. ACM.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

[5] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS*, pages 85–96. ACM, 2009.

[6] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

[7] Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1498–1503, Munich, Germany, March 2008.

[8] Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[9] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *LCPC*, pages 235–250, 2006.

[10] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, New York, NY, USA, 2009. ACM.

[11] Baolin Shao, Nalini Vasudevan, and Stephen A. Edwards. Compositional deadlock detection for rendezvous communication. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 59–66, Grenoble, France, October 2009.

[12] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[13] Nalini Vasudevan and Stephen A. Edwards. Static deadlock detection for the SHIM concurrent language. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 49–58, Anaheim, California, June 2008.

[14] Nalini Vasudevan and Stephen A. Edwards. Celling SHIM: Compiling deterministic concurrency to a heterogeneous multicore. In *Proceedings of the Symposium on Applied Computing (SAC)*, volume III, pages 1626–1631, Honolulu, Hawaii, March 2009.

[15] Nalini Vasudevan and Stephen A. Edwards. A determinizing compiler. In *Programming Languages Design and Implementation (PLDI) - Fun Ideas and Thoughts Session*, Dublin, Ireland, June 2009.

[16] Nalini Vasudevan and Stephen A. Edwards. Determinism should ensure deadlock-freedom. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, Berkeley, California, June 2010.

[17] Nalini Vasudevan, Satnam Singh, and Stephen A. Edwards. A deterministic multi-way rendezvous library for Haskell. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, Miami, Florida, April 2008.