D^2C: A Deterministic, Deadlock-free Concurrent Programming Model

Nalini Vasudevan
Columbia University
New York, NY
naliniv@cs.columbia.edu

***Abstract*—The advent of multicore processors has made concurrent programming models mandatory. However, most concurrent programming models come with a repertoire of problems. The two major ones are non-determinism and deadlocks. By determinism, we mean the output behavior of the program is independent of the interleaving caused by the schedule and depends only on the input behavior. A few concurrent models provide deterministic behavior by providing constructs like barriers and locks that impose additional synchronization, but the incorrect usage of these constructs leads to problems like deadlocks.**

In this paper, we propose D^2C, a new programming model that guarantees the two desirable properties of concurrency - determinism and deadlock-freedom. Any program in this model will be deterministic; the output of the program will solely depend on the input and not on the interleaving of the tasks in the program. Additionally, the model cannot introduce deadlocks. We prove the correctness of our model and evaluate it with a set of examples.

***Keywords*-Determinism, Deadlock-Freedom**

I. Introduction

Non-deterministic behavior 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 and both take x by reference. 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.

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

Figure 1. A non-deterministic concurrent program

Such non-determinism makes debugging very hard because unwanted behavior is rarely reproducible. Re-running a non-deterministic program on the same input usually does not produce the same behavior. Debugging then becomes a nightmare.

By contrast, all sequential programming languages (e.g., C) are deterministic: they produce the same output given the same input. This helps programmers by making it easy to verify a program because if a program produces the desired result for an input during testing, it will do so reliably.

Concurrent models based on atomic transactions and locks are race free but are not deterministic. For instance, protecting x by a lock in Figure 1 will still produce non-deterministic output. Concurrent software languages are generally based on these models and use traditional shared memory, locks, and condition variables (e.g., pthreads or Java). They are non-deterministic because the output of a program may depend on such things as the operating system's scheduling policy, the relative execution rates of parallel processors, and other things outside the application programmer's control. Not only does this demand a programmer consider the effects of these things when designing the program, it also means testing can only say a program may behave correctly on certain inputs, not that it will.

We agree with Bocchino et al. [2] that the programming environment should ensure input-output determinism. By determinism, we mean that the program's output should only depend on the input, and not on the environment (operating system schedule, processor, cache etc.). There are a number of models and tools that aid determinism. We discuss them in Section VII.

While determinism and deterministic concurrent models are interesting, they give rise to a number of problems. For example, if the tasks do not synchronize in the right order defined by the synchronization protocol, we obtain a deadlock. A deadlock is a situation when two or more tasks are indefinitely A simple classic example of a deadlock is *lock(p); lock(q);* by one task and *lock(q); lock(p);* by another. waiting for each other to finish. Deadlocks are frustrating and generally hard to manually detect during run-time.

One way to address these problems is to build a deterministic, deadlock-free concurrent programming model and D^2C is an instance. Any application written in this model

```
1  void f(clocked int ref a) {
2      /* x is 1 */
3      a <- 3;
4      /* x is still 1 */
5      next; /* The reduction operator is applied */
6      /* x is 8 */
7  }
8
9  void g(clocked int ref b) {
10     int local;
11     /* x is 1 */
12     b <- 5;
13     /* x is still 1 */
14     local = b; /* local is 1 */
15     next; /* The reduction operator is applied */
16     /* x is 8 */
17     local = b; /* local is 8 */
18 }
19
20 void h (clocked int ref c) {
21      /* x is still 1 */
22      next;
23     /* x is 8 */
24 }
25
26 main() {
27      clocked(+) int x = 1;
28      /* If there are multiple writers, reduce
29       using the + reduction operator */
30      spawn clocked(x) f(x) ;
31      spawn clocked(x) g(x);
32      h(x);
33      next;
34      /* x is 8 */
35 }
```

Figure 2. Example of a program written in our model

will guarantee to give the same output for a given input and will never deadlock. It is important to note that we do not guarantee termination. We do not try to solve the halting problem here - our model is applicable only for terminating programs.

We start by discussing our model in Section II and a few examples in Section III. We then provide a proof of correctness (determinism and deadlock-freedom) in Section IV. We provide a basic implementation of the runtime in Section V. We then evaluate our model by experimenting on a set of examples in Section VI. We compare our work with related work in Section VII and finally conclude with future work in Section VIII.

II. D^2C: Our model

Non-determinism arises when multiple tasks concurrently modify a shared variable. Our programming model is a modification to Cilk - we allow multiple tasks to write to a shared variable concurrently but in a synchronized fashion and 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 clocked variables are reduced using the reduction operator.

In the example in Figure 2, the reduction operator is $+$ because x is clocked with a reduction operator $+$ in Line 27. 0 is the initial value applied to the reduction operator while reducing. Every task that uses a clocked variable should explicitly share that variable in the $spawn$ statement. For instance, in Figure 2, each of f and g , because they explicitly have $clocked(x))$ at the spawn statement. The main task that executes h is clocked on x because main declares it. At the $next$ statement, every clocked variable in the task advances its phase and the values offered by clocked variables in the previous phase are reduced and used in the current phase.

The statement $a \leftarrow 3$, does a delayed write to variable a, which is a reference to x, i.e., a value 3 is offered to the next phase of a. When the task calls $next$, the task advances its phase, forcing the value 3 to be seen by other tasks.

Therefore after the $next$ statement, the values offered by different tasks are reduced and henceforth 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.

A task may share multiple variables. The $next$ statement is a conjunctive barrier on all clocked variables. A task holding a clocked variable waits for all other tasks that is also holding the same clocked variable to either call *next* or to terminate.

A single task may offer multiple values to the same variable in one phase. These values are also reduced using the commutative associative operator along with the values offered by other tasks. For instance, in Figure 3, after the execution of $next$ in task f, the value of x is 4;

The programming model is deterministic because writes to a particular variable are made visible only in the next phase. Also, the model is deadlock-free: a task A waiting on another task B at the *next* statement will eventually proceed, because task B at some point will call *next* or terminate.

A phase is either separated by two $next$s or separated by the creation of the clocked variable and a $next$. If no values are offered in a given phase, then the value from the previous phase is maintained.

If the programmer does not declare a reduction operator, and there are multiple offers in the same phase, then at the 'next' statement following the phase, the offers are rejected and the value from the previous phase is maintained. If there

```
1  void f(clocked int ref a) {
2      a <− 1;
3      a <− 1;
4      a <− 1;
5      next;
6      /∗ x is 4 ∗/
7  }
8
9  void g(clocked int ref b) {
10     int local;
11     a <− 1;
12     next; /∗ The reduction operator is applied ∗/
13     /∗ x is 4 ∗/
14 }
15
16 main() {
17     clocked(+) int x = 0;
18     /∗ If there are multiple writers, reduce
19     using the + reduction operator ∗/
20     spawn clocked(x) f(x) ;
21     g(x);
22   /∗ x is 4 ∗/
23 }
```

Figure 3. Multiple offers in a single phase

```
1  void f(int value, clocked int ref b[M]) {
2      int bucket = value % M;
3      b[bucket] = 1;
4  }
5
6  main() {
7      const int a[N] = {...};
8      clocked((+) b[M] ;
9      for (i = 0; i < N; i++)
10         spawn clocked(b) f(a[i], b);
11     next; /∗ Reduction happens here ∗/ ;
12
13 }
```

Figure 4. Histogram example in D^2C

is just one offer, then the following '$next$' statement makes this new value visible to other tasks.

III. Examples in D^2C

We illustrate our model with a few examples: histogram, pipeline and merge-sort

A. Histogram

The program in Figure 4 calculated the frequencies of values from array a into array b. Array b is written concurrently. Therefore it should be clocked. Multiple tasks may write 1 to the same location in b, but these values are reduced using the $+$ operator in Line 11. After the $next$ statement, array b will contain the correctly computed histogram of array a.

B. Pipeline

The Pipeline example in Figure 5 creates 3 stages in parallel. *Stage1* gets the input, *stage2* increments it by 1 and passes it to *stage3*. *Stage3*, again increments it by one and prints the value. Every task reads the value from the current phase and offers the value to the next phase. For instance *stage2* reads a from the current phase and offers a new value to b. But this new value of b is seen by *stage3* only in the next phase. The pipeline example does not need a reduction operator because only one task is writing to a clocked variable at any phase. The pipeline application is a classic example of a program that has multiple phases but with single write at each phase.

```
1  void stage1(clocked int ref a) {
2      int i = 0;
3      for (;; i++) {
4          a <− i;
5          next;
6      }
7
8  }
9
10 void stage2 (clocked int ref a, clocked int ref b) {
11     next; /∗ Skip first clock ∗/
12     for (;;) {
13         b <− a + 1; /∗ b is 1, 2, 3... ∗/
14         next;
15     }
16
17
18 void stage3 (clocked int ref b) {
19     int c;
20     next; /∗ Skip first clock ∗/
21     next; /∗ Skip second clock ∗/
22     for (;;) {
23         c <− b + 1;
24         print(c); /∗ Prints 2, 3, 4, 5 .... ∗/
25         next;
26     }
27
28 }
29
30 main() {
31     clocked int a, b, c;
32     spawn clocked(a) stage1 (a);
33     spawn clocked(b) stage2 (a, b);
34     stage3 (b);
35 }
```

Figure 5. Pipeline example in D^2C

C. Merge Sort

The merge-sort (Figure 6) is a classic example of red-black computation but in place. At every stage, the merging task reads from the current phase and offers the new (merged) values to the next phase, does eliminating the need of two explicit arrays.

The $sort$ function spawns two $sort$ functions on the sub-arrays and waits for them to finish at the $next$ statement. The $merge$ function takes the two sub-arrays and merges them. The merged values are offered to the next phase. After executing the $merge$ function, the executing task

returns to its parent function ($sort$) and terminates itself. The parent $sort$ that spawned this task waits until all its children terminate at the $next$ statement in Line 14. After this statement the values offered at the previous phase are shifted to the current phase.

```
1  void sort (clocked int a[], const int start, const int end) {
2      /* first half */
3      const int fStart = start;
4      const int fEnd = start + (end − start)/2;
5      /* second half */
6      const int sStart = fEnd + 1;
7      const int sEnd = end;
8
9      if (start == end)
10         return;
11
12     spawn clocked(a) sort (a, fStart, fEnd); /* Sort first half */
13     sort (a, sStart, sEnd); /* Sort second half */
14     next; /* Wait for the sub−arrays to be sorted */ ;
15     merge(a, fStart, fEnd, sStart, sEnd);
16 }
17
18 void merge (clocked int ref a[], const int fStart,
19     const int fEnd, const int sStart, const int sEnd) {
20
21     int x = fStart;
22     int y = sStart;
23     int z = fStart;
24     const int size = (sEnd − fStart) + 1;
25
26     /* Read from a and offer to the next phase of a */
27
28     while(x <= fEnd && y <= sEnd) {
29         if(a(x) < a(y))
30             a(z++) = a(x++);
31         else
32             a(z++) = a(y++);
33     }
34     while(x <= fEnd) {
35         a(z++) = a(x++);
36     }
37     while(y <= sEnd) {
38         a(z++) = a(y++);
39     }
40 }
41
42 main {
43     const int N = 100;
44     clocked int a[N];
45     /* Initialize a below */
46     ..
47     next; /* Move the initialized values to the next phase */
48     sort(a, 0, N − 1);
49     next; /* Move the offered sorted array to the next phase */
50     print (a) ;
51   }
52 }
```

Figure 6. Merge Sort in D^2C

IV. Proof of Correctness

In this section, we prove that our model guarantees the two desirable properties of concurrency: determinism and deadlock-freedom.

A. Determinism

A task always reads from the current phase and writes to the next phase. Therefore, there are no read-write conflicts. At the $next$ statement, all concurrent writes are reduced using an associative, commutative reduction operator. If there is no declared operator but multiple writes, the writes are ignored. This ensures write-write conflict freedom.

Whenever a clocked variable is created, the creating $task$ owns that variable. Whenever the clocked variable is used by a *spawn*, the spawned task also owns the variable. At the $next$ statement, the task waits for all threads that share clocked variables with it to either call *next* or terminate.

A task will always synchronize with every task that has already spawned and shares variables with it. Synchronization is deterministic. We prove this by contradiction. Consider two tasks t_i and t_j that share variable x. Suppose a task t_i synchronizes with all tasks that share variable x but an already spawned t_j. This implies that t_i is not aware that t_j has been spawned. Since t_j uses variable x, t_j's parent t_p should also use variable x. t_p is aware that t_j has already spawned because it is t_j's immediate parent. Therefore t_p will synchronize with t_j thereafter. When t_i synchronizes with t_p's $next$, t_p will synchronize with t_j, forcing t_i to synchronize with t_j, and therefore contradicting the statement that t_i will miss t_j. This follows from the fact that a $spawn$ can happen in serial with a $next$ statement and not concurrently with it. Secondly, parent should also clock x if it spawns a child that clocks x, unless x is declared in the child.

B. Deadlock-freedom

The only statement that a task can wait on is the $next$ statement. Suppose a task t_i waits on the $next$ statement for task t_j. t_i waits until t_j either executes $next$ or terminates. Every task will either call $next$ or terminate and therefore t_i can never deadlock. The $next$ statement behaves like a conjunctive join on all clocked variables.

Suppose t_i waits on t_j, t_j waits on t_k, and so on and finally t_n calls $next$ to wait for t_i, then there is a cycle in the graph. But t_n realizes that t_i has already called $next$, and therefore it does not wait on t_i, breaking the cycle.

We do not allow Cilk's $sync$ statements in our model because it can cause deadlocks with $next$ statements. But a $sync$ statement can be implemented using a $next$ statement as follows:

```
1 clocked void s;
2 spawn clocked(s) f(s);
3 spawn clocked(s) g(s);
4 next;
```

The *next* statement forces all tasks that hold s to either call *next* or terminate. If the body of f and g does not use $next$, then the role of the $next$ statement is the same as a $sync$ statement.

V. Implementation

We implemented our model in the X10 programming language [3]. 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 task in Cilk; 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.

In our implementation, we do not support activities at multiple places; we assume all activities run in a single place - something similar to a shared memory system. We force all shared variables to be clocked. This forces race-freedom. If a shared variable is not clocked, the compiler throws an error.

During runtime, we maintain two states for each variable, *write state* and *read state*. If the clocked variable is read, it is simple read from the *read state*. If the clock variable is written, it is written but atomically to the *write state*. The value in the *write state* and the value to be written are reduced using the reduction operator, and this newly obtained value is written back to the *write state*. This access and modification to the write state is done atomically to ensure determinism during concurrent writes by different activities. Whenever, a *next* is called, we swap the references to the *read state* and *write state*. The *write state* of the previous phase is now the *read state* and vice-versa. We also clear the *write state* of the current phase. We ensure determinism by duplicating states. The amount of memory is independent on the number of tasks or the interleavings, but is only twice of the original program.

Every time a value is written to a shared variable, a lock is obtained. This was a major bottleneck in many of our benchmarks. To overcome this problem, we maintained a copy of the shared variable for every thread created in the program. Since every thread has its own copy, a lock is not necessary. At the $next$ statement, we reduce the values offered by the different threads. With this technique, we achieved about $3-4\%$ improvement in performance. Now, the memory does not depend on the interleavings but is directly proportional to the number of tasks.

VI. Experimental Results

To test the performance of our model, we ran a number of examples on a 1.6 GHz Quad-Core Intel Xeon (E5310) server running Linux kernel 2.6.20 with SMP (Fedora Core 6). The processor "chip" actually consists of two dice, each containing a pair of processor cores. Each core has a 32 KB L1 instruction and a 32 KB L1 data cache, and each die has a 4 MB of shared L2 cache shared between the two cores.

We tested our model with real applications. Figure 7 shows the results. We measured the deterministic implementation of the applications with the original implementation. A bar with value below 1 indicates that the deterministic version ran slower than the original version.

Each of the applications created 4 threads. The All-Reduce 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 Monte-Carlo 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, Ray-Trace, Lu-Fact, SparseMatMul and Series programs are JGF benchmarks. The Ray-tracer 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 Lu-Fact 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 [4] 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 performance degradation 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 the implementation maintains a duplicate copy to eliminate read-write conflicts. Hence, the deterministic version performed better. The Java and C++ versions did about the same.

To measure the performance of the synchronizing $next$ statement, which is the real bottleneck of our model, we created a shared variable. Each task updates calls $next$ 500 times, and between every two $next$s, it updates the shared variable. This forces each task to synchronize with every other task 500 times. All tasks do exactly the same job.

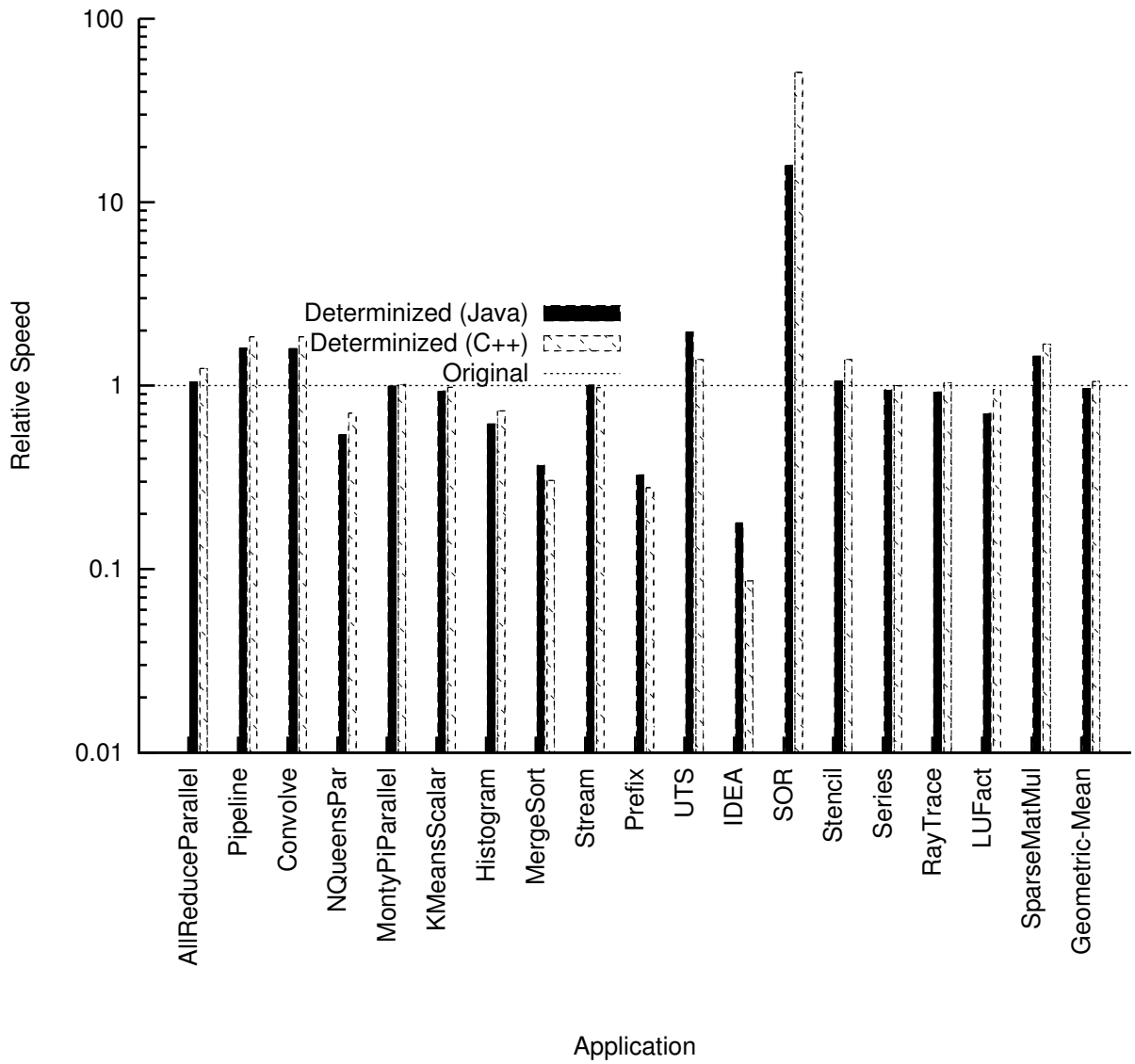

Figure 7. Relative performance of the determinized applications on a quad core machine

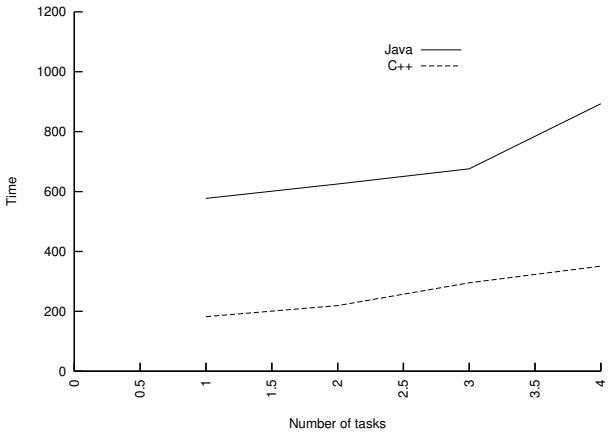

Figure 8. Performance of $next$ statement with varying number of tasks

Figure 8 shows the output. The x-axis represents the number of tasks and the y-axis is the time.

In a perfect scalable system, if we add a new task to the system, then we expect the speed to remain the same (assuming the number of tasks is equal to the number of cores). Our system is not perfectly scalable, as expected, therefore we see a curve rather than a horizontal line for both the C++ and Java versions in Figure 8.

VII. Related Work

A number of groups are working on a similar problem. In this section, we review some of the related work and compare them with ours.

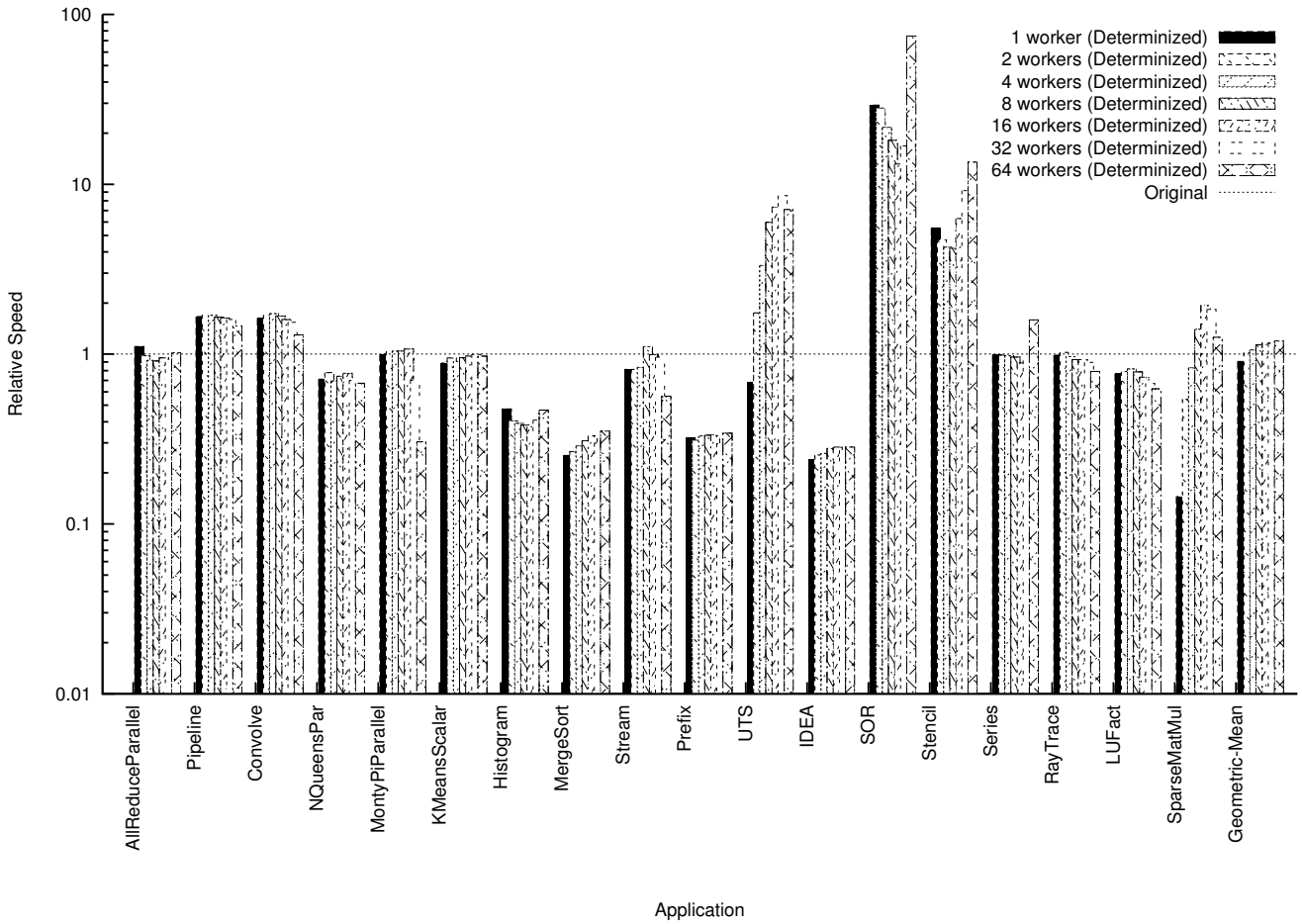

Figure 9. Experimental results with 64 cores

A. Determinizing Tools

A number of tools provide determinism. For example, Kendo is a purely software system that deterministically multi-threads concurrent applications. Kendo [5] ensures a deterministic order of all lock acquisitions for a given program input.

Kendo comes with three shortcomings. It operates completely at runtime, and there is considerable performance penalty. Secondly, if we have the sequence *lock(A); lock (B)* in one thread and *lock(B); lock(A)* in another thread, a deterministic ordering of locks may still deadlock. Thirdly, the tool operates only when shared data is protected by locks.

Software Transactional Memory (STM) [6] is an alternative to locks: a thread completes modifications to shared memory without regard for what other threads might be doing. At the end of the transaction, it validates and commits if the validation was successful, otherwise it rolls back and re-executes the transaction. STM mechanisms avoid races but do not solve the non-determinism problem.

Berger's Grace[7] is a run-time tool that is based on STM. If there is a conflict during commit, the threads are committed in a particular sequential order (determined by the order The problem with Grace is that it incurs a lot of run-time overhead. This dissertation partially solves this overhead problem by addressing the issue at compile-time and thereby reducing a considerable amount of run-time overhead.

Like Grace, Determinator[8] is another tool that allows parallel processes to execute as long as they do not share resources. If they do share resources and the accesses are unsafe, then the operating throws an exception (a page fault).

Cored-Det [9], based on DMP [10] uses a deterministic token that is passed among all threads. A thread to modify a shared variable must first wait for the token and for all threads to block on that token. DMP is hardware based. Although, deadlocks may be avoided, we believe this setting is non-distributed because it forces all threads to synchronize and therefore leads to a considerable performance penalty. In the D^2C setting, only threads that share a particular channel must synchronize on that channel; other threads can run independently.

Deterministic replay systems [11], [12] facilitate debug-

ging of concurrent programs to produce repeatable behavior. They are based on record/replay systems. The system replays a specific behavior (such as thread interleaving) of a concurrent program based on records. The primary purpose of replay systems is debugging; they do not guarantee determinism. They incur a high runtime overhead and are input dependent. For every new input, a new set of records is generally maintained.

Like replay systems, Burmin and Sen [13] provide a framework for checking determinism for multi-threaded programs. Their tool does not introduce deadlocks, but 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. Our goal is to guarantee determinism at compile time – given a program, it will generate the same output for a given input.

B. Programming Models

synchronous programming languages like Esterel are completely deterministic. An Esterel program executes in clock steps and the outputs are conceptually synchronous with its inputs. It is a finite state language that is easy to verify formally. An Esterel program is susceptible to causalities. Causalities are similar to deadlocks, but can be easily detected at compile-time. The problem with synchronous models is that they do not perform well. To out knowledge, most Esterel compilers generate sequential code and there are hardly any compilers that generate concurrent code off Esterel.

SHIM [14], [15] is also a deterministic concurrent programming language, but the improper use of its constructs leads to problems such as deadlocks i.e., a SHIM program may be susceptible to deadlocks. Any program written in our model is always deadlock-free. Secondly, SHIM allows only a single task to write at any phase; we allow multiply writes.

Apart from SHIM, there are a few programming models and languages that provide explicit determinism. StreamIt [16], for example is a synchronous dataflow language that provides determinism. It has simple static verification techniques for deadlock and buffer-overflow. However, StreamIt is a strict subset of SHIM and StreamIt's design limits it to a small class of streaming applications.

In contrast, Cilk [1] is a non-deterministic language that it covers a larger class of applications. It is C based and the programmer must explicitly ask for parallelism using the *spawn* and the *sync* constructs. Cilk is definitely more expressive than D^2C. However, Cilk allows data races. Figure 1, for example, is a non-deterministic concurrent program in Cilk. Explicit techniques [17] are required for checking data races in Cilk programs.

X10 [3], [18] is another language that adopts the Cilk model. X10 is non-deterministic - but we were able to modify the compiler and runtime with a very few changes to incorporate our model in it.

The reduction operator in D^2C model is very similar to map-reduce. However, are a few differences. We can have multiple phases of map-reduce in D^2C. We have used map-reduce as one of the means to obtain determinism. Secondly, not always do we require an operator in D^2C. The D^2C model is also applicable for applications with single writes.

C. Type Systems and Verifiers

Finally, type and effect systems like DPJ [19] have been designed for deterministic parallel programming to see if memory locations overlap. Our technique is more explicit. In general, type systems require the programmer to manually annotate the program. Our model can also be implemented using annotations in existing programming languages - we in fact annotated the X10 programming language.

Martin Vechev's tool [20] finds determinacy bugs in loops that run parallel bodies. It analyzes array references and indices to ensure that there are no read-write and

VIII. Conclusions

We have presented a concurrent model of computing that addresses the two major problems of concurrency: non-determinism and deadlocks. We have proved the correctness of our model. We have evaluated our model and shown that our model works reasonably well on many examples. Our model is merely a simple construct that gives determinism, flexible enough to write a range of codes, and is implementable with little overhead.

Our current runtime implementation is very basic; We wish to optimize the implementation especially for array data structures.

All clocked variables are created on the stack. We do not deal with heap data structures because it introduces aliasing problems. We wish to improve our implementation by allowing variables to be created on the heap. However, we did not find the lack of heap as a huge bottleneck. We could build all the applications in Section VI without using the heap.

Currently, we force all shared variables to be clocked. We would like to improve our model by incorporating intelligent static analysis to reduce runtime overhead.

We have implemented our model as annotations in an existing programming language, X10. We also wish to explore the possibilities of implementing such a model as a library.

Lastly, the challenge still remains to verify if the reduction operator is commutative and associative especially with user defined operators. However, this problem is simplified because operators cannot have concurrent activities in it. Nate Clarke's work [21] on commutative analysis for instance uses randomized testing, but does not completely verify the commutative property.

REFERENCES

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, California, Jul. 1995, pp. 207–216.

[2] R. L. B. Jr., V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *HOTPAR '09: USENIX Workshop on Hot Topics in Parallelism*, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2008.4536264

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, 2005.

[4] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: An unbalanced tree search benchmark," in *LCPC*, 2006, pp. 235–250.

[5] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2009, pp. 97–108.

[6] N. Shavit and D. Touitou, "Software transactional memory," in *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1995, pp. 204–213.

[7] E. D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: safe multithreaded programming for c/c++," in *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM, 2009, pp. 81–96.

[8] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, "Efficient system-enforced deterministic parallelism," in *Proceedings of Operating System Design and Implemetation (OSDI)*, Vancouver, BC, Canada, Oct. 2010, pp. 193–206.

[9] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "Coredet: a compiler and runtime system for deterministic multithreaded execution," *SIGARCH Comput. Archit. News*, vol. 38, pp. 53–64, March 2010. [Online]. Available: http://doi.acm.org/10.1145/1735970.1736029

[10] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "Dmp: deterministic shared memory multiprocessing." in *ASPLOS*. ACM, 2009, pp. 85–96. [Online]. Available: http://dblp.uni-trier.de/db/conf/asplos/asplos2009.html#DeviettiLCO09

[11] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*. New York, NY, USA: ACM, 1998, pp. 48–59.

[12] G. Altekar and I. Stoica, "Odr: output-deterministic replay for multicore debugging," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 193–206.

[13] J. Burnim and K. 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 software engineering on European software engineering conference and foundations of software engineering symposium*. New York, NY, USA: ACM, 2009, pp. 3–12.

[14] S. A. Edwards and O. Tardieu, "SHIM: A deterministic model for heterogeneous embedded systems," in *Proceedings of the International Conference on Embedded Software (Emsoft)*, Jersey City, New Jersey, Sep. 2005, pp. 37–44. [Online]. Available: http://doi.acm.org/10.1145/1086228.1086277

[15] O. Tardieu and S. A. Edwards, "Scheduling-independent threads and exceptions in SHIM," in *Proceedings of the International Conference on Embedded Software (Emsoft)*, Seoul, Korea, Oct. 2006, pp. 142–151. [Online]. Available: http://doi.acm.org/10.1145/1176887.1176908

[16] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe, "StreamIt: A compiler for streaming applications," Dec. 2001, mIT-LCS Technical Memo TM-622, Cambridge, MA. [Online]. Available: citeseer.ist.psu.edu/article/thies01streamit.html

[17] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark, "Detecting data races in cilk programs that use locks," in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 298–309. [Online]. Available: http://doi.acm.org/10.1145/277651.277696

[18] V. A. Saraswat, V. Sarkar, and C. von Praun, "X10: concurrent programming for modern architectures," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2007, pp. 271–271.

[19] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. 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*. New York, NY, USA: ACM, 2009, pp. 97–116.

[20] M. Vechev, E. Yahav, R. Raman, and V. Sarkar, "Automatic verification of determinism for structured parallel programs," in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Cousot and M. Martel, Eds., vol. 6337. Springer Berlin / Heidelberg, 2011, pp. 455–471.

[21] F. Aleen and N. Clark, "Commutativity analysis for software parallelization: letting program transformations see the big picture," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508273