Deterministic Galois: On-demand, Portable and Parameterless *

Donald Nguyen Andrew Lenharth Keshav Pingali
The University of Texas at Austin, Texas, USA
{ddn@cs, lenharth@ices, pingali@cs}.utexas.edu

Abstract

Non-determinism in program execution can make program development and debugging difficult. In this paper, we argue that solutions to this problem should be *on-demand*, *portable* and *parameterless*. On-demand means that the programming model should permit the writing of non-deterministic programs since these programs often perform better than deterministic programs for the same problem. Portable means that the program should produce the same answer even if it is run on different machines. Parameterless means that if there are machine-dependent scheduling parameters that must be tuned for good performance, they must not affect the output.

Although many solutions for deterministic program execution have been proposed in the literature, they fall short along one or more of these dimensions. To remedy this, we propose a new approach, based on the Galois programming model, in which (i) the programming model permits the writing of non-deterministic programs and (ii) the runtime system executes these programs deterministically if needed. Evaluation of this approach on a collection of benchmarks from the PARSEC, PBBS, and Lonestar suites shows that it delivers deterministic execution with substantially less overhead than other systems in the literature.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Concurrent programming structures

Keywords Deterministic scheduling; Irregular programs; Multicore processors

* The work presented in this paper has been supported by NSF grants CCF 1337281, CCF 1218568, ACI 1216701, and CNS 1064956. Donald Nguyen was supported by a DOE Sandia Fellowship.

ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.

ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541964

1. Introduction

Non-determinism in program execution can make program development and debugging difficult. A number of systems for executing parallel programs deterministically have been proposed recently.

Some solutions make programs deterministic by *construction* by restricting the programming model to prevent application developers from writing non-deterministic programs; data-parallel, stream and functional programming models are well-known examples [7, 9, 11, 27].

Other solutions provide determinism by *scheduling*: they permit developers to write non-deterministic programs but restrict the possible schedules of operations in these programs to ensure deterministic execution. These restrictions can be enforced at different levels of the execution stack including hardware [16, 17, 21], OS [2, 4], compiler [3], and software runtime [24, 25].

Ideally, a deterministic parallel system would provide the following three features.

- **On-demand determinism.** It should be possible to turn deterministic execution on and off without much effort. As we show in Section 5, deterministic execution often imposes a substantial runtime overhead, particularly for parallel programs with fine-grain tasks. This overhead may be acceptable in some cases, but it should be possible to turn off determinism when desired.
- **Portability.** The output of a deterministic program should be the same regardless of the machine that it runs on. At the very least, this means that the output should not depend on the number of executing threads. Portability ensures that programs enjoy the benefits of determinism even when moving between machines.
- **Parameter-freedom.** If there are scheduling parameters that must be tuned to achieve good performance, they should not affect the output state. Since optimal values for such parameters vary by machine, such scheduling parameters hinder portability by providing an incentive for producing different results on different machines.

No existing system for deterministic parallel execution currently provides on-demand determinism, portability and parameter-freedom.

Programming models such as DPJ [9], revisions with deterministic merges [11], StreamIt [27], and nested data-parallel programs [8] are deterministic by construction, but they are incompatible with on-demand determinism. Other programming models like Grace [5] are not deterministic by construction, but the output of a program may depend on the number of threads used to execute the program. There are deterministic programming models that have "escapes" that allow non-deterministic programs to be written, such as DPJ with non-determinism [10] and PBBS [7], but there is no convenient method for producing deterministic executions on demand from such programs.

When determinism is provided by scheduling, it is usually not portable. In hardware systems like RCDC [17] and Calvin [21], the number of threads is a fundamental part of program representation, and any deterministic execution is always with respect to a particular number of threads. At the OS and compiler level, it is possible to virtualize the concept of threads to achieve portability, but current systems such as Determinator [2], CoreDet [3] and dOS [4] do not do this. Runtime approaches are replacements for non-deterministic program libraries. Both DThreads [24] and Kendo [25] model the behavior of pthreads, an explicitly threaded library, and are not portable.

Finally, some of the above systems have user-tunable scheduling parameters that affect execution performance *and* output: examples are RCDC, CoreDet, Kendo, and PBBS.

In this paper, we present a deterministic parallel system that is on-demand, portable and parameter-free. It takes high-level, non-deterministic programs written in the Galois programming model [26] as input. These programs can be executed non-deterministically with the Galois runtime, or they can be executed deterministically using a technique that we call *deterministic interference graph (DIG) scheduling*. The application program does not have to change when switching between non-deterministic and deterministic scheduling since the desired scheduler is specified through a command-line parameter.

To evaluate the quality of our deterministic programs, we compare their performance with handwritten deterministic parallel programs from the PBBS benchmark suite [7]. Our results show a median performance of 0.62X compared to the handwritten PBBS implementations. To highlight the importance of on-demand determinism, we compare the performance of non-deterministic Galois programs with that of deterministic PBBS programs. The non-deterministic programs achieve a median speedup of 2.4X over the PBBS implementations.

In addition, we compare our DIG scheduling approach with a prior deterministic system, CoreDet [3], on non-deterministic PBBS programs. CoreDet can make any threaded program deterministic. The CoreDet execution achieves scaling for only one out of the four applications. This is because unlike the PARSEC benchmarks [6] commonly used to evaluate deterministic parallel systems, our benchmarks have much more synchronization and smaller task sizes, which severely taxes systems with high scheduling overheads.

```
1  foreach Task t in P:
2    atomic:
3      t() // execute task
4      enqueue(S(t)) // enqueue new tasks, if any
```

(a) Non-deterministic program.

```
1  foreach Task t in P:
2    if writeMarks(0, id(t), L(t)):
3      // all writes successful
4      t()
5      enqueue(S(t))
6    else:
7      enqueue(t)
8    writeMarks(id(t), 0, L(t))
```

(b) Scheduling non-deterministic programs.

Figure 1: Non-deterministic program and scheduling (see Figure 3 for auxiliary function definitions).

The rest of this paper is organized as follows. In Section 2, we describe the non-deterministic Galois programming model and provide a high-level overview of scheduling with interference graphs. In Section 3, we discuss the implementation of this approach in the Galois system. Section 4 summarizes the benchmarks used in the evaluation. In Section 5, we measure the performance of various deterministic and non-deterministic programs, as well as the performance of the programs with CoreDet. Section 6 summarizes related work, and Section 7 concludes the paper.

2. Galois programming model

The programming model used is an abstract version of the Galois programming model for unordered algorithms [26]. There are no explicitly parallel constructs like threads and locks; instead, parallelism is specified implicitly through the use of the iterator shown in Figure 1a. Key features are the following.

- P is a pool of tasks that can be performed in *any* order. The program terminates when all tasks have been executed.
- When task t is completed, it may create a set of new tasks $S(t)$, which are added to the task pool. Task t is the *parent* of the tasks it creates; the transitive closure of the parent relation is called the *ancestor* relation.
- Each task performs computation and reads and writes shared-memory locations. The set of locations read $R(t)$ and written $W(t)$ by a task t is said to constitute its *neighborhood*, which is denoted by $L(t) = R(t) \cup W(t)$.

Tasks are required to be *cautious*: that is, a task must read all of the locations in its neighborhood before it can write to any of them.

- A *conflict* occurs between tasks t_1 and t_2 if (i) neither task is an ancestor of the other, and (ii) one of them writes to the neighborhood of the other ($W(t_1) \cap L(t_2) \neq \emptyset$). If there can be no conflicts between tasks, parallel execution is straightforward since the program is a generalized data-parallel loop in which the iteration range can grow dynamically. In the presence of conflicts, a correct parallel schedule for the program should be serializable: it must appear as if all tasks were performed atomically in some order that respects the ancestor relation.

Although neighborhoods can be defined as sets of concrete memory locations, it is better to define them as sets of *abstract* memory locations; for example, for a graph algorithm, these abstract locations might correspond to graph elements such as nodes and edges of the graph, rather than to the concrete memory locations implementing this abstract data type. The Galois runtime system implements synchronization by associating abstract locks or marks with abstract locations (§2.1).

2.1 Non-deterministic scheduling

In most programs, neighborhoods of tasks are not known statically. One parallelization strategy is to use speculative execution: a free thread selects an arbitrary task from the task pool P and executes it, rolling back the task if a conflict with another concurrently executing task is detected. Tracking of neighborhoods can be done using a transactional-memory-like approach over abstract memory locations [19, 20].

For programming models with cautious tasks, conflict detection and correction can be done using much lighter weight mechanisms because the synchronization problem reduces to the well-known dining philosopher's problem [12]. Conceptually, each abstract location can be *acquired* by an owner. The execution of a task can be divided into two phases: in the first phase, a task reads locations but does not write to any of them, acquiring ownership of these locations, and in the second phase, the task writes to some locations, but it does not write to any location that it did not read in the first phase. The point between the first and second phase is called the *failsafe point*. For cautious tasks, conflicts are detected in the first phase, and rollback is implemented simply by releasing ownership of all locations. Once the failsafe point has been crossed, global data structures can be updated in place without the need for backup copies of modified data.

Figure 1b implements this idea for non-deterministic task scheduling. For each abstract memory location l, we maintain a mark location Mark(l). Each task t has a unique id id(t), and there is an id 0 that is distinct from all other ids. For this implementation, ids need only be unique. For the deterministic scheduler below (§3), ids must also have a total order, and 0 must be less than any other id. Mark locations initially contain the value 0. The scheduler chooses a task t and tries to acquire ownership of all the locations in the neighborhood of that task, using compare-and-set instructions for example. If successful, the task executes and adds new tasks $S(t)$ to the pool. If unsuccessful, there is a conflict, and the scheduler adds t back to the pool for re-execution. In either case, any marks acquired are reset back to 0.

3. Deterministic scheduling

In this section, we describe our implementation of deterministic scheduling. It is based on finding independent sets in an implicitly constructed *interference graph* of tasks. We first describe the high-level idea (§3.1), then its implementation (§3.2), and finally some important optimizations (§3.3).

3.1 DIG scheduling

Definition 1. *Given a set of tasks P, an* interference graph *for P is an undirected graph $G_P = (V_P, E_P)$ in which there is a distinct node in V_P representing each task in P, and there is an undirected edge $(v_1, v_2) \in E_P$ if the tasks represented by v_1 and v_2 have a conflict.*

The interference graph for a set of tasks can be built by executing each task up to its failsafe point while tracking its neighborhood and putting a conflict edge between two tasks if their neighborhoods overlap. This is a conservative approach since it puts a conflict edge between two tasks even if they both read a location that neither of them writes to. Program analysis can be used to determine conflicts more accurately; for the purposes of this paper, any conservative interference graph is adequate.

Interference graphs can be used to schedule tasks as follows. The tasks in the task pool P are executed in *rounds*. In each round, the scheduler performs the following activities:

- *inspect:* build an interference graph G_P for the tasks in P,
- *select:* find an *independent set* $\mathcal{I}$ of nodes in G_P and remove the corresponding tasks from P, and
- *execute:* execute the tasks in $\mathcal{I}$ in parallel, adding any newly created tasks to P.

Scheduling is completed when all tasks have been executed. During the *select* phase, it is desirable but not necessary to find a maximal independent set of nodes in the graph.

A subtle point is that the interference graph must in general be rebuilt from scratch each round since the neighborhood of a task is relative to the global state, which is modified by tasks in the *execute* phase.

There is one enhancement to this basic scheme that is useful for reducing the overhead of interference graph construction. Note that the scheduling strategy works correctly even if, in each round, the interference graph is constructed only for a subset of tasks in the pool; the remaining tasks are simply delayed to later rounds. This *windowing* scheme can reduce the overhead of interference graph construction

```
1  Tasks cur, next, todo
2  todo = P;
3  while ||todo|| > 0:
4    // order tasks in todo deterministically
5    next = sort(todo)
6    todo = {}
7    // execute sorted tasks
8    while ||next|| > 0: // for each round...
9      calculateWindow()
10     barrier
11     // get prefix of size window from next
12     cur,next = getWindowOfTasks(next)
13     // compute neighborhoods of tasks in cur
14     inspect(cur)
15     barrier
16     // execute successful tasks, move
17     // failed tasks to next, and add any
18     // newly created tasks to todo set
19     todo = todo ∪ selectAndExec(cur, next)
20     barrier
```

Figure 2: Deterministic scheduler.

when the number of tasks is much larger than the number of threads. For the windowed scheduler to be deterministic, we must also ensure the following in each round: (i) tasks for the current window are chosen deterministically from the task pool, and (ii) during the select phase, the independent set of nodes is chosen deterministically.

3.2 Implementation of DIG scheduling

Figure 2 shows the pseudocode for the implementation of deterministic scheduling; auxiliary functions are shown in Figure 3. In the pseudocode, **doall** indicates a loop whose iterations are run in parallel. Instead of explicitly building an interference graph, this code directly finds an independent set of tasks by using marks on locations.

A summary of what the scheduler does is the following. The task set *todo* is initialized to the initial set of tasks P. These tasks are ordered deterministically to form a sequence *next*. This sequence of tasks is executed over several rounds; in each round, a prefix *cur* of tasks in *next* is tried for execution. Some of these will succeed and others may fail. Tasks created by successful tasks are added to *todo*; these are executed after *next* becomes empty. Failed tasks are added back to *next* and retried in later rounds. Execution terminates when *todo* and *next* become empty.

Some important implementation details are the following.

The *inspect* operation uses *writeMarksMax* to mark the neighborhood of a task, stealing ownership of neighborhood locations from tasks with lower ids. While the mark on a given location may be updated non-deterministically depending on how the tasks in *cur* are scheduled, the final mark will be the same regardless of the order in which tasks were processed in the inspect phase. This is because computing the maximum (or minimum) element of a set with a total order is deterministic (the set in question is the set of

```
1  Tasks sort(Tasks todo):
2    // sort tasks in set todo
3
4  void inspect(Tasks cur):
5    doall t in cur:
6      writeMarksMax(id(t), L(t))
7
8  Tasks selectAndExec(Tasks cur, Tasks next):
9    Tasks newWork = {}
10   doall t in cur:
11     if readMarks(L(t)) = {id(t)}:
12       // all reads equal id so execute
13       execute task t
14       // add new work if any to newWork
15       newWork = newWork ∪ S(t)
16     else:
17       next = next ∪ t
18     writeMarks(id(t), 0, L(t))
19   return newWork
20
21
22 bool writeMarks(Id expected, Id id, Set<Loc> locs):
23   for loc in locs:
24     atomic:
25       if Mark(loc) == expected:
26         Mark(loc) = id
27       else: return false
28   return true
29
30 void writeMarksMax(Id id, Set<Loc> locs):
31   for loc in locs:
32     atomic:
33       if Mark(loc) < id:
34         Mark(loc) = id
35
36 Set<Id> readMarks(Set<Loc> locs):
37   // return set of ids in mark locations
```

Figure 3: Auxiliary functions for deterministic scheduler.

ids of the tasks that read or wrote to a particular location in the current round). The cumulative effect of the writeMarkMax operations implicitly creates an interference graph. An edge in the interference graph exists between tasks t_1 and t_2 if $\text{id}(t_2) \in \text{readMarks}(L(t_1))$.

One important difference between writeMarks and writeMarksMax is that writeMarks can fail early if it cannot update a mark location, but in order to be deterministic, writeMarksMax must attempt to update all mark locations even if it failed to update some of them. If a task skips some mark locations, it changes the set that writeMarksMax is computing the maximum of, and if the mark locations skipped depend on a scheduling choice, then the resulting maximums are non-deterministic.

In the second phase, the scheduler selects and executes an independent set of tasks (line 19 in Figure 2 and function selectAndExec in Figure 3). A task is selected if all of its neighborhood locations are still marked with its id at the end of the inspection phase. Tasks selected this way form an independent set in the interference graph. This set is unique because of the total order on ids. If any of the neighborhood

locations of a task does not contain its id, the task is not part of the independent set, and it is placed in the *next* set to be executed in a future round. In either case, the marks written by a task are cleared in preparation for the next round.

Execution continues in rounds until there are no tasks left in *next*. If there are no tasks in the *todo* set, the scheduler terminates; otherwise these tasks are moved to *next*, and execution continues. Note that in each round, the task in *cur* with maximum id is guaranteed to execute, so each round executes at least one task.

Before enqueued tasks can be scheduled, they must be assigned a unique id. The assignment of ids must also be deterministic. Ids are assigned as follows. The initial tasks are given ids based on the iteration order of the C++ iterator that contains the tasks. When task t creates task u, the scheduler stores with task u the id of the task that created it $\mathrm{id}(t)$ and a number k indicating whether it was the first, second, third, etc. task created by t. In the sort function, tasks are sorted lexicographically based on the pair $(\mathrm{id}(t), k)$, and the scheduler uses the position in the total order defined by the sort as the id for the new tasks.

The performance of this scheduler depends critically on the window size, so we implemented an adaptive algorithm that grows and shrinks the window size each round depending on the number of tasks that successfully committed in the previous round. The getWindowOfTasks and calculateWindow functions in Figure 2 implement this functionality. The calculateWindow function computes the window size for the current round based on the fraction of tasks that committed in the previous round. If the commit ratio is less than some target threshold, the next window size is scaled down proportionally. If the commit ratio is above the threshold, the window size is doubled. The getWindowOfTasks function simply returns this prefix of tasks in *cur* and postpones the remainder to *next*. Since the number of tasks that commit in a round is independent of the number of executing threads, this heuristic is portable across machines.

To implement the deterministic marking scheme in the Galois system, we made two changes to the existing system.

First, the default mark values in the Galois system are not ordered. We modified the marking code to keep track of the id of a task and to use that value appropriately when writing mark values.

Second, neighborhoods are not explicitly maintained by the Galois system. Marks are acquired incrementally during execution via user code calls to a data structure library. The only way to get the neighborhood of a task is to execute the task and observe which marks are acquired. To implement the inspect phase, we simply execute a task, which, by its normal execution, marks locations in its neighborhood. When the task reaches its failsafe point (the first write to a global location), it immediately returns. To implement the selectAndExec phase, we re-execute the task from the beginning, and instead of writing marks, we check whether the marks that we would have written match the values that have been written. This implements line 11 of Figure 3. If a task reads a mark value that is not its id, we go to line 17.

This baseline implementation is sufficient to deterministically schedule any program written in the programming model of Figure 1a.

3.3 Optimizations

The baseline deterministic scheduler described in Section 3.2 contains several inefficiencies, which we now address.

First, it redundantly executes the prefix of a task up to its failsafe point when a task is selected and executed. A more efficient method would be to suspend execution of a task at the failsafe point during the inspect phase and to resume execution in the commit phase. On resumption, the task must check that all the mark values still match its id (Figure 3 line 11). The capability to pause and resume execution can be achieved generally using additional threads or creating continuations. In our optimized implementation, we use a more ad-hoc approach, which simulates the effect of forming a continuation without implementing a full compiler transform. We provide a library function that allows users to allocate objects in the inspect phase which can be recalled during the commit phase. Programmers can use this functionality to manually achieve the same effect as task suspend and resume.

To make sure that resumed tasks are valid to commit, we make a small change to the protocol in the inspect phase. Instead of just writing the maximum mark value, a task t checks if the previous value of the mark location is not 0 and not $\mathrm{id}(t)$; if so, by writing its mark, task t will prevent the task u that corresponds to the current mark value from committing. Normally, task u detects this case when the scheduler executes line 11, or in the case of the baseline scheduler, when task u is executed a second time. When using the continuation optimization, t is now responsible for preventing u from executing. It does this by writing to a flag variable that u checks before resuming execution.

Second, the performance of the scheduler is very sensitive to initial task order. Applications that exploit temporal locality execute tasks with overlapping neighborhoods close in time. This typically translates to those tasks being close together in iteration order, which, in the baseline scheduler implementation, means that they typically will be executed in the same round, where they will certainly conflict with each other. This leads to the perverse situation where the scheduler needs to reduce locality to improve performance. We address this issue by assuming that tasks placed close together in iteration order have high locality and place those tasks in separate rounds if possible.

Third, the cost of sorting enqueued tasks can be large relative to the application time. There is a common special case where a task enqueues tasks, but those tasks are drawn from a fixed set of tasks. In this case, tasks can be assigned unique ids before parallel execution, and the programmer

can pass these ids to the scheduler, which uses them directly instead of generating new ids via the sort function.

3.4 Comparison of non-deterministic and deterministic schedulers

Compared to the non-deterministic scheduling in Section 2.1, DIG scheduling adds several overheads, whose performance impact we evaluate in Section 5.

- As seen in Figures 1b and 2, the deterministic scheduler executes many more instructions.
- The deterministic scheduler introduces a concept of rounds that is not present in the original program. These rounds are implemented using global synchronization. Rounds extend the critical path length of a program because the scheduler cannot proceed to the next round until all of the tasks are processed for the current round.
- The scheduler executes tasks according to a particular schedule, but that schedule may not be the best performing one among possible program schedules.
- The execution of a task is broken into two parts, the inspect phase and the execution phase, separated by a barrier. The memory locations accessed during the inspect phase of a task are very likely to be accessed by the execution phase of the same task, but under DIG scheduling, these two phases are temporally separated by a factor that is a function of number of tasks attempted during a round, which is typically very large. Conversely, increasing locality by reducing the number tasks attempted in a round, increases the number of rounds executed, which increases the critical path length of the program.

4. Experimental setup

4.1 Applications

The benchmarks in our study are drawn from three different sources: the PARSEC (v2.1) benchmark suite [6], the problem based benchmark suite (PBBS) (v0.1) [7], and the Lonestar (v2.1.5) benchmark suite [22].

PARSEC The PARSEC benchmark suite has been used in previous evaluations of deterministic scheduling [3, 17, 24]. It contains twelve applications or kernels. Most are parallelized using the pthreads library. We chose the three benchmarks that have OpenMP implementations: **blackscholes**, **bodytrack** and **freqmine**.

PBBS The PBBS programs [7] are organized by problem, and each problem has one or more solution programs, at least one of which is deterministic. There are a total of sixteen problems, but many of these programs are data-parallel or nested data-parallel, and their performance depends largely on factors like good load balancing, which is not the subject of this paper. We therefore excluded them from our study. We chose deterministic programs that solved the four remaining problems: breadth-first search (**bfs**), Delaunay triangulation (**dt**), Delaunay mesh refinement (**dmr**), and maximal independent set (**mis**). We exclude maximal matching because of its similarity to maximal independent set. In these codes, determinism is ensured by application-specific techniques customized to each application, and they typically involve bulk-synchronous execution in rounds. The PBBS maximal independent set program is data-parallel, but we have included it in our study for comparison with a non-deterministic maximal independent set program that exists in the Lonestar suite.

Lonestar From the Lonestar benchmark suite, we selected four programs that solve the same problems as those we included from PBBS, using the same algorithms, and an implementation of the preflow-push algorithm (**pfp**) that uses the global relabeling heuristic to improve convergence [13]. We automatically generate deterministic implementations of all Lonestar programs by applying the DIG scheduling of Section 3, including the optimizations of Section 3.3.

There is one small difference between the PBBS and Lonestar implementations of Delaunay triangulation (dt). The algorithmic complexity of Delaunay triangulation depends on the order in which points are inserted, and random insertion order has been shown to be optimal [14]. In the PBBS implementation, points are randomized offline. In the Lonestar implementation, points are reordered online using the biased randomized insertion order algorithm [1]. For comparison purposes, we do not include the reordering time in either implementation.

As mentioned above, the Lonestar maximal independent set program is non-deterministic while the PBBS version is data-parallel.

Application variants In the experimental results, the variant **g-n** denotes the original non-deterministic Lonestar application, and the deterministic variant generated from DIG scheduling is called **g-d**. The variant **PBBS** denotes the PBBS version of the application.

4.2 Data-sets

For the PARSEC benchmarks, we use the simlarge inputs for blackscholes and freqmine and the native input for bodytrack.

The performance of the PBBS and Lonestar benchmarks can vary significantly with the type of input. In our experience, the behavior across random inputs for an application is largely similar, so we choose a single representative input for each application. These inputs are largely drawn from the evaluation of Blelloch et al. [7]. For bfs, we use a random graph of 10 million nodes where each node is connected to five randomly selected nodes. For dmr, we use a Delaunay triangulated mesh of 2.5 million randomly selected points from the unit square. For dt, we use 10 million points randomly selected from a unit square. For mis, we use the same input as bfs. For pfp, we use a random graph of 2^{23} nodes with each node connected to 4 random neighbors.

4.3 Platforms

We use three machines in our evaluations and take the average of at least three runs for each application/machine/thread-count combination. The three machines are (i) **m4x10**, a machine running Ubuntu Linux 10.04 LTS 64-bit (Linux 2.6.32) with four ten-core Intel Xeon E7-4860 (2.27 GHz) processors; (ii) **m4x6**, a machine running Ubuntu Linux 10.04 LTS 64-bit (Linux 2.6.32) with four six-core Intel Xeon E7540 (2.0 GHz) processors; and (iii) **numa8x4**, an SGI UV machine (ccNUMA) running SuSE Enterprise 11 SP1 64-bit (Linux 2.6.32.24) with eight four-core Intel E7520 (1.87 GHz) processors. The processors of numa8x4 are divided into blades of two processors each and enclosures of two blades each. Inter-blade communication uses SGI NUMALink 5.

We compile the PBBS and Lonestar programs with icc version 12.1 with the -O3 optimization flag. For the PBBS programs, we use the Cilk runtime to manage and load balance threads. For the Lonestar programs, we use the Galois runtime system.

5. Evaluation

In Section 5.1, we describe applications characteristics useful for understanding the performance results.

In Section 5.2, we compare the performance of non-deterministic programs with their performance when executed with CoreDet [3], a system that provides determinism by scheduling. Because the C++ language primitives used in the Lonestar programs are not supported by CoreDet, we used non-deterministic versions of the PBBS programs as our representative non-deterministic benchmarks. We show that with CoreDet, the non-deterministic PBBS programs do not perform well and have a median slowdown of 3.7X (min: 1.3X, max: 55X) compared to running without CoreDet. These experiments show that systems like CoreDet that provide determinism through deterministic thread scheduling are not suitable for irregular applications, which have relatively fine-grain tasks.

In Section 5.3, we compare the performance of non-deterministic Galois programs (g-n), generated deterministic implementations of these Galois programs (g-d), and handwritten deterministic PBBS programs for the same problems. Overall, our results show that at the maximum number of threads on each machine, (i) g-n variants achieve a median improvement of 4.2X compared to g-d, (ii) g-n variants are 2.4X faster than the PBBS variants, and (iii) g-d variants are only 0.62X slower than the PBBS variants.

These results show that the automatically generated deterministic Galois programs are comparable in performance to the handwritten PBBS programs and that there is a significant performance penalty for deterministic execution. A study with performance counters in Section 5.4 reveals that, for the most part, non-deterministic programs perform better than deterministic ones because they exploit more locality.

			$p = 1$		$p = 40$	
	Variant	Rounds	Abort Ratio	Tasks per μs	Abort Ratio	Tasks per μs
bfs	g-d	1700	0.08	0.45	0.08	9.76
bfs	g-n	0	0	1.32	0	39.92
bfs	pbbs	11	0	1.24	0	24.27
dmr	g-d	1287	0.11	0.13	0.11	2.53
dmr	g-n	0	0	0.26	< 0.01	8.98
dmr	pbbs	1165	0.03	0.18	0.03	2.90
dt	g-d	35213	0.27	0.12	0.27	1.78
dt	g-n	0	0	0.24	< 0.01	7.47
dt	pbbs	1330	0.10	0.11	0.10	2.48
mis	g-d	100	0.08	0.77	0.08	21.05
mis	g-n	0	0	3.98	< 0.01	79.69
mis	pbbs	29	0.05	14.59	0.05	143.12
pfp	g-d	21047	0.04	0.26	0.04	2.58
pfp	g-n	0	0	0.67	< 0.01	14.99

Figure 4: Abort ratio and task execution rates on machine m4x10.

		$p = 1$		$p = 40$	
	Variant	Count	Rate	Count	Rate
mis	pbbs	< 1	< 0.01	< 1	< 0.01
freqmine		< 1	< 0.01	< 1	< 0.01
bodytrack		< 1	< 0.01	15	0.07
dt	pbbs	1445	0.55	1522	0.66
blackscholes		< 1	< 0.01	48	0.77
bfs	pbbs	7191	1.24	7162	1.36
dmr	pbbs	2360	1.00	2634	1.37
pfp	g-n	4622	2.24	1519	27.57
dmr	g-n	707	1.59	583	36.21
dt	g-n	1376	3.10	920	79.94
mis	g-n	19292	10.27	4628	100.17

Figure 5: Atomic updates by application measured by binary instrumentation on machine m4x10. Variant g-d omitted. Count is atomic updates per million instructions executed. Rate is atomic updates per microsecond.

5.1 Application characteristics

The first set of measurements is concerned with the parallel behavior of applications, such as task granularity, frequency of synchronization, and how often inter-task conflicts occur during execution. These metrics are of interest for the following reasons.

Previous evaluations of deterministic scheduling have focused mostly on applications with coarse-grain tasks that

communicate relatively infrequently. Deterministic scheduling for these kinds of applications can be supported using relatively heavyweight mechanisms since the overhead of the system is a small fraction of the overall execution time. However, these mechanisms may not be useful for applications with very lightweight tasks that communicate frequently. On a shared-memory system, the concept of communication is less well-defined compared to a distributed system, but one approximation is the number of atomic updates an application performs. Figures 4 and 5 show task execution rates, abort ratios, and atomic update rates for our applications on 1 thread and 40 threads on machine m4x10. For the deterministic variants, the number of rounds is also shown. For PBBS variants, this is the number of bulk-synchronous rounds of the handwritten deterministic scheduling.

First, we see that the PBBS and Lonestar benchmarks have very fine-grain tasks. For example, the g-n version of dmr, running on one thread, commits 0.26 tasks per microsecond (see Figure 4), which translates to 3.8 microseconds per task (this is the parallel version of the code with synchronization, running on one thread), which is on the order of a thousand cycles. On 40 threads, this parallel program commits roughly 9 tasks per microsecond, which translates to a throughput of roughly 0.11 microseconds per task.

Second, we see that the abort ratios of the g-n variants of all applications are essentially zero even at 40 threads. Conflicts between tasks in the non-deterministic variants are very rare: this is because there are a large number of tasks compared to the number of threads. The deterministic variants g-d and PBBS have larger abort ratios because in each round, the number of tasks whose neighborhoods are inspected is typically larger than the number of threads. Conflicts can also happen with only one thread when two tasks with overlapping neighborhoods are inspected in the same round.

Third, the PARSEC benchmarks—blackscholes, bodytrack and freqmine, which are frequently used to evaluate deterministic schedulers—have orders of magnitude fewer atomic updates than the irregular algorithms of the PBBS and Lonestar suites (see Figure 5). For example, blackscholes at 40 threads performs atomic updates at a rate of about 1 update per microsecond, while the mis g-n variant performs atomic updates at the rate of 100 updates per microsecond.

These qualitative differences in application characteristics significantly impact the design of deterministic schedulers, as we show in the following sections.

5.2 Deterministic thread scheduling

In this section, we present performance results from using CoreDet, a deterministic thread scheduler, on our benchmark applications. Unlike DIG scheduling, CoreDet runs on unmodified pthread programs.

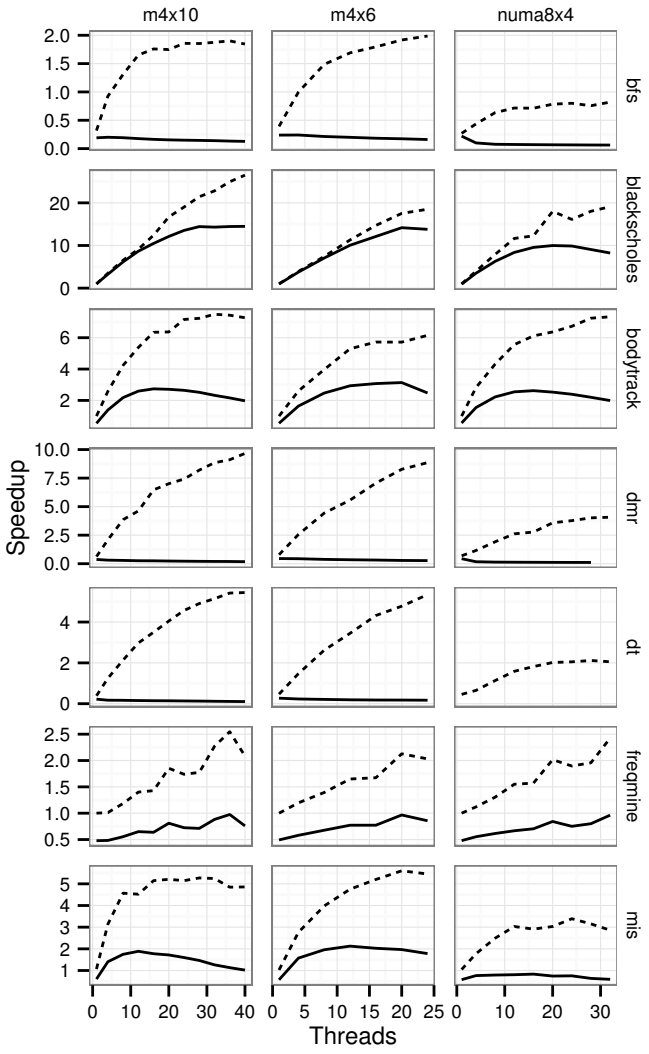

Figure 6: Speedup with (solid lines) and without (dotted lines) CoreDet system on non-deterministic programs. Speedup baselines are in Figure 8. Some dmr and dt runs on numa8x4 timed out after 10 minutes.

Ideally, we would like to run the PARSEC and g-n non-deterministic programs with CoreDet to make them deterministic. Unfortunately, the CoreDet compiler is based on the older LLVM 2.6 compiler, and it is unable to compile any of the g-n programs. To get around this problem, we exploit the fact that bfs, dmr and dt in PBBS are deterministic implementations of non-deterministic algorithms. To make them non-deterministic, we transform the programs by hand to match the non-deterministic program pattern shown in Figure 1 and run these with CoreDet. We leave the mis benchmark as a data-parallel program.

We use the CoreDet system in low-overhead, synchronization-only mode, which reduces the system to an implementation of the Kendo algorithm [25] and requires all synchronization between threads to use the pthread library.

For all our CoreDet comparisons, we replace calls to the OpenMP or Cilk runtimes with calls to a simple pthread-only runtime. The simple pthread-only runtime likely introduces minor inefficiencies compared to the more optimized Cilk and OpenMP runtimes, but they are dwarfed by the overheads of CoreDet. We used the LLVM 2.6 compiler with the -O3 optimization flag to compile programs.

Figure 6 summarizes our results using the CoreDet system to make PARSEC and modified PBBS programs deterministic. CoreDet works well for blackscholes: the performance with CoreDet is almost the same as without CoreDet for a small number of threads. As the number of threads increases, the gap between using CoreDet and not using CoreDet increases, hinting at a serialization bottleneck in the deterministic scheduler. The bodytrack and freqmine applications show more limited speedups. For the modified PBBS programs, the performance with CoreDet is poor except for mis, the data-parallel code. The bfs, dmr and dt applications perform substantially more synchronization than the PARSEC applications and the mis code (§5.1). Overall, at the maximum number of threads on each machine, the benchmarks in this suite experience a median slowdown of 3.7X (min: 1.3X, max: 55X) compared to non-CoreDet runs.

Although this evaluation uses CoreDet, the other deterministic thread schedulers that we are familiar with such as Kendo and DThreads have similar scheduling algorithms and differ mainly in how they deal with racy data accesses, which none of the modified PBBS programs have.

These results make the case that a different approach than deterministic thread scheduling is needed to handle applications that perform orders of magnitude more synchronization than more conventional programs like the PARSEC benchmarks.

5.3 End-to-end performance of g-n, g-d and PBBS

Figure 7 shows the speedups of g-n, g-d and PBBS relative to the best performing serial implementations shown in Figure 8. For bfs, the baseline is the highly optimized code of Schardl and Leiserson that uses data structures customized to the bfs problem [23]. For preflow-push, we use the highly optimized hi_pr implementation from Goldberg and Tarjan [18]. For the other benchmarks, the best performing versions that we were able to find were from the benchmark suites considered in this paper.

Figure 7 shows that the best performing variant overall is g-n with a median improvement of 2.4X over corresponding PBBS programs at the maximum number of threads on each machine. The benefit is largest on the numa8x4 machine where the scalability of the PBBS variant is particularly poor, but there are positive benefits for almost all non-deterministic variants. Figure 7 suggests that there are also significant scalability advantages to the non-deterministic variants compared to the deterministic ones. The g-n variants are able to achieve at least a 15X speedup on m4x10 for four of the five applications.

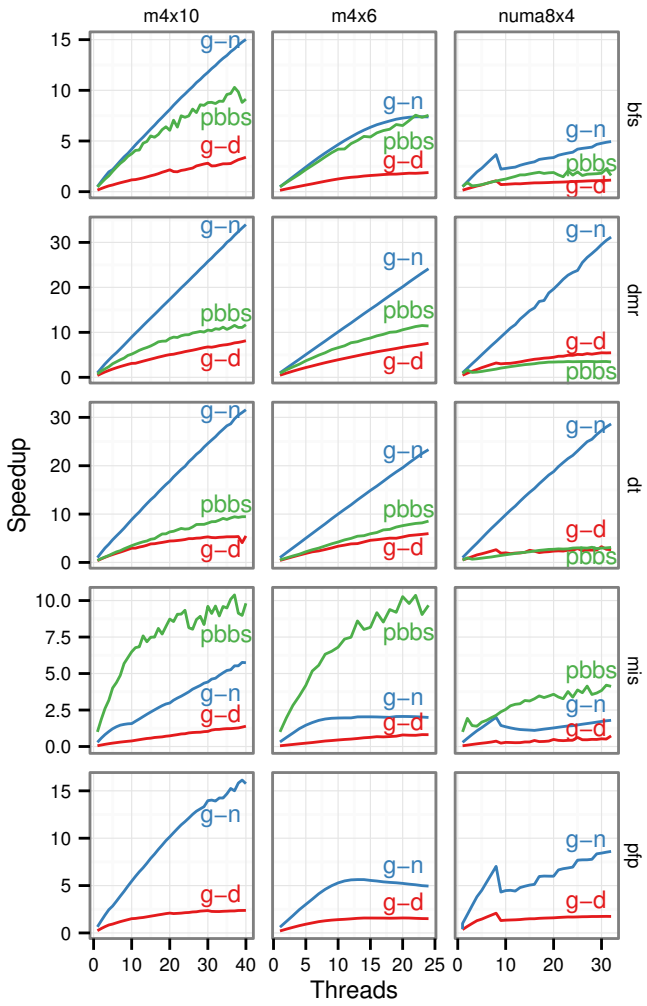

Figure 7: Speedup for selected deterministic and non-deterministic variants. Speedup baselines are in Figure 8.

	Machine	Var.	Time (s)		Machine	Var.	Time (s)
bfs	m4x10	cilk	3.76	dt	m4x10	g-nd	42.35
bfs	m4x6	cilk	4.36	dt	m4x6	g-nd	56.37
bfs	numa8x4	cilk	4.85	dt	numa8x4	g-nd	61.15
bs	m4x10		8.42	fm	m4x10		7.95
bs	m4x6		11.27	fm	m4x6		10.57
bs	numa8x4		12.01	fm	numa8x4		11.33
bt	m4x10		164.84	mis	m4x10	pbbs	0.72
bt	m4x6		216.31	mis	m4x6	pbbs	0.90
bt	numa8x4		249.07	mis	numa8x4	pbbs	0.91
dmr	m4x10	g-nd	44.48	pfp	m4x10	hi_pr	13.64
dmr	m4x6	g-nd	60.04	pfp	m4x6	hi_pr	14.64
dmr	numa8x4	g-nd	63.29	pfp	numa8x4	g-nd	26.17

Figure 8: Baseline times in seconds for speedup calculations (bs: blackscholes, bt: bodytrack, fm: freqmine). These are the best times for any variant with one thread. Cilk is a parallel bfs code [23]. hi_pr is a sequential implementation of pfp [18].

The main outlier in these results is the behavior of mis. As mentioned above, the PBBS variant of mis is a data-parallel program, and its execution characteristics are significantly different than the g-n or g-d variants. The main conclusion from this benchmark is that if one has a deterministic algorithm for some problem, it may be better to use that algorithm rather than deterministically schedule a non-deterministic algorithm for the same problem. However, as we will see in Section 5.4, the PBBS variant is more sensitive to input ordering than g-n.

The sharp drop in performance at eight threads in some of the numa8x4 runs is caused by inter-NUMA node communication. Runs of eight threads or less are scheduled to run on a single NUMA node. Runs with more than eight threads use more than one NUMA node, and remote memory accesses are significantly more expensive than local memory accesses. The g-n variants for dmr and dt are able to tolerate the transition to inter-node communication due to locality optimizations in the Galois runtime system. Exploiting locality may be difficult in deterministically scheduled programs, as we describe in Section 5.4.

We conclude that for irregular applications, determinism comes at a significant price in performance, even if the determinism is obtained through hand-optimized, application-specific code.

We note that a previous study by Blelloch et al. [7] reached the opposite conclusion; for example, they found that the deterministic PBBS version of dt was substantially faster than the non-deterministic dt program in the Lonestar suite. Unfortunately, their study did not ensure that the same algorithm was used for a given problem; in particular, the dt algorithm in the PBBS suite is different (and more efficient) than the one that was used in the Lonestar suite at the time their study was performed. For the study in this paper, we reimplemented in the Lonestar suite the dt algorithm used in PBBS, so in our measurements, the performance differences between non-deterministic and deterministic programs for a given problem are not entangled with algorithmic differences.

DIG scheduling vs. determinism by construction How good is the general-purpose deterministic scheduler described in Figure 2 compared to the application-specific, hand-optimized deterministic code in the PBBS programs? Figure 9 shows the performance for different variants relative to the same baseline, the PBBS variant.

Across all machines and benchmarks and at the maximum number of threads (I_{max}), the median performance of the g-d variant relative to PBBS is only 0.62X. If we drop mis from the set of benchmarks, the median performance is 0.70X.

We conclude that the general-purpose deterministic scheduler described in Figure 2 provides reasonable performance compared to application-specific, hand-optimized determinism by construction code, although there is room for improvement.

For dmr and dt, the PBBS variants correspond to a hand-written version of DIG scheduling of the g-nd variants. We believe the performance difference between the g-d and PBBS variants is largely due to the application-specific implementation of resuming tasks and the handtuned window selection policy used in PBBS. It would be interesting to study whether a compiler could optimize the generic code of Figure 2 and provide tuned implementations for each application.

Impact of continuation optimization The g-d variants use the continuation optimization described in Section 3.3, which requires some user input to form proper continuations (this transformation could be done by a compiler, but we have not implemented this yet). To weigh the effect of this optimization, we measure the performance of programs without this optimization. Overall, the continuation optimization provides a median improvement of 1.14X for our deterministic programs. As seen in Figure 10, the continuation optimization provides a significant improvement only for the relatively more complicated dmr and dt programs.

5.4 Determinism and locality

In Section 3.4, we describe how DIG scheduling can decrease the performance of an application relative to its non-deterministic implementation. In this section, we quantify two of those costs. First, DIG scheduling can make it more difficult to exploit locality, and second, DIG scheduling can reduce existing locality.

Performance counter measurements for locality As mentioned in Section 3.4, DIG scheduling decreases locality by splitting a task, which might have significant intra-task locality, into two phases well-separated in time. We can quantify the impact of this transformation by measuring performance counter information about memory-level events.

Figure 11 gives the number of data requests satisfied from DRAM for the g-n, g-d, and PBBS versions of our applications. The non-deterministic variants typically have far fewer samples than the deterministic ones, but is the change in samples enough to explain the difference in performance?

One way to answer this question is to see how the observed data fits a simple model of performance. Let efficiency be speedup normalized by the number of threads. One simple model is that there is a linear relationship between the change in efficiency and the change of some performance counter. Symbolically, let eff_{var} and PC_{var} be the efficiency and performance counter value, respectively, of some application variant with some number of threads on a machine, and let eff_{ref} and PC_{ref} be the likewise for a particular reference variant of the same application with the same number of threads. We would like to know how well the following linear model fits the observed data

$$\text{eff}_{\text{var}} = B_0 + B_1(\text{PC}_{\text{ref}}/\text{PC}_{\text{var}})\text{eff}_{\text{ref}}.$$

		m4x10				m4x6				numa8x4			
	Variant	Mean	Max	I_1	I_{max}	Mean	Max	I_1	I_{max}	Mean	Max	I_1	I_{max}
bfs	g-n	1.28	1.68	1.07	1.64	1.10	1.20	1.00	0.98	2.23	3.48	1.00	3.09
bfs	g-d	0.31	0.37	0.33	0.37	0.29	0.32	0.28	0.25	0.61	1.03	0.30	0.71
bfs	pbbs	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
dmr	g-n	2.12	2.99	1.39	2.90	1.63	2.11	1.11	2.11	5.45	9.12	1.18	9.12
dmr	g-d	0.62	0.71	0.57	0.70	0.60	0.66	0.52	0.66	1.38	1.75	0.55	1.59
dmr	pbbs	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
dt	g-n	2.81	3.34	2.34	3.34	2.43	2.76	1.92	2.73	6.70	9.30	2.07	9.26
dt	g-d	0.72	0.95	0.87	0.58	0.77	0.89	0.79	0.70	1.15	2.08	1.11	0.86
dt	pbbs	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
mis	g-n	0.40	0.64	0.29	0.59	0.27	0.35	0.31	0.21	0.48	0.94	0.28	0.44
mis	g-d	0.09	0.15	0.05	0.14	0.07	0.09	0.05	0.08	0.12	0.18	0.05	0.18
mis	pbbs	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 9: Performance of variants relative to PBBS variant. Let $\text{time}_{\text{PBBS}}(p)$ and $\text{time}_{\text{var}}(p)$ be the times for variant PBBS and var respectively with p threads. The performance number shown is $\text{time}_{\text{PBBS}}(p)/\text{time}_{\text{var}}(p)$. I_1 and I_{max} show performance at 1 and the maximum number of threads respectively.

		m4x10			
	Variant	Mean	Max	I_1	I_{max}
bfs	g-d	0.31	0.37	0.33	0.37
bfs	without	0.27	0.32	0.30	0.32
dmr	g-d	0.62	0.71	0.57	0.70
dmr	without	0.48	0.54	0.43	0.52
dt	g-d	0.72	0.95	0.87	0.58
dt	without	0.56	0.71	0.68	0.49
mis	g-d	0.09	0.15	0.05	0.14
mis	without	0.08	0.13	0.05	0.12

Figure 10: Performance without continuation optimization relative to PBBS variant on machine m4x10. I_1 and I_{max} show performance at 1 and the maximum number of threads respectively.

Fitting the above linear model to our observed data on machine m4x10 reveals that the change in DRAM accesses significantly predicts the change in performance, $\beta = 0.35$, $t(108) = 16.8$, $p < 0.001$. The change in this performance counter also explained a significant portion of the variance in change of performance, $R^2 = 0.72$, $F(1, 108) = 282$, $p < 0.001$. There are performance counters that are more highly correlated ($R^2 \geq 0.75$), clock cycles for instance, but that relationship is trivial.

Inter-task locality Figure 12 shows how the performance of mis can vary on the same input, depending on whether the input is randomized or sorted. The input graph is a 2D mesh which is ordered by sorting nodes according to a space-filling curve. Figure 12(ordered) shows that the g-n variant is able to effectively exploit the locality in the input data and obtain far better performance than the g-d or PBBS variants. When the input graph is randomized, there is no locality to be exploited, and the PBBS version performs slightly better than the g-n version.

Non-deterministic programs can more readily exploit both intra-task and inter-task locality. The execution of a single task is not divided into phases separated in time, so they can exploit intra-task locality better. Furthermore, locality in the input data, which leads to inter-task locality, is easier to exploit.

6. Related work

The majority of deterministic parallel systems work by executing tasks in rounds and deterministically resolving conflicts when two tasks access the same resource in a round. A common way to resolve conflicts is to buffer updates privately and then deterministically merge updates to form the new state for the next round. Hardware systems like RCDC [17] and Calvin [21] work this way, as well as runtime replacements like DThreads [24] and Kendo [25], compiler-based systems like CoreDet [3], OS systems like dOS [4] and Determinator [2], and some parallel programming models like Grace [5].

In systems that are deterministic by construction such as DPJ [9], nested data-parallel programs [8], stream programs [27] and commutativity-based techniques [7, 11], tasks have no conflicts. However, it is not possible to write non-deterministic programs in these approaches. Bocchino et al. have shown how to extend DPJ to compose safely with non-deterministic programs [10], but there is as yet no system to make the resulting program deterministic on demand.

For round-based systems, an important characteristic of the system is how tasks are determined. The hardware systems and Kendo use the number of instructions executed (or a similar proxy) to divide sequences of instructions into tasks. Depending on how conflicts are detected, task boundaries may also have to be formed at every memory fence, synchronization instruction, store buffer completion, etc. RCDC and the bounded mode of Calvin form tasks based on when the store buffer is full, which is a micro-architectural event. This means that programs may not be deterministic across different processor implementations. CoreDet, Kendo and the unbounded mode of Calvin form tasks based on the number of executed instructions, so their results should be the same between processors.

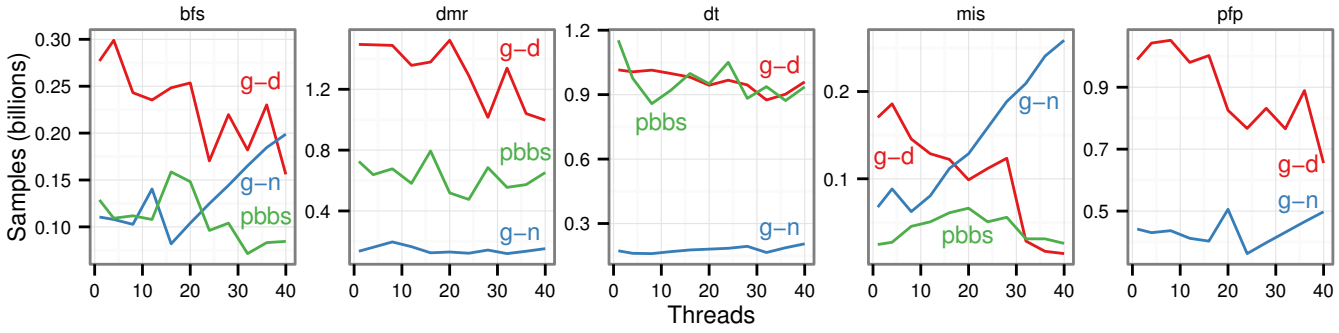

Figure 11: Samples of DRAM access performance counter on machine m4x10.

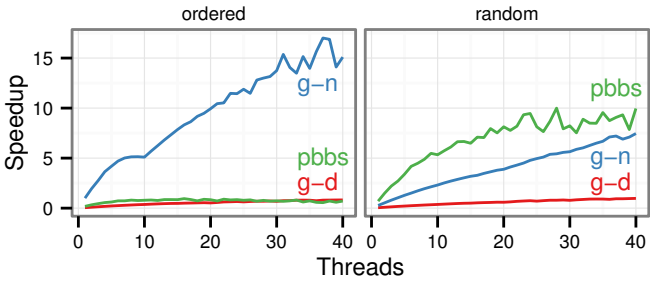

Figure 12: Effect of ordering input for mis. Speedup is relative to the best variant with one thread among the two inputs (g-n, 4.3 seconds). The ordered input is a graph of a 2D mesh. The nodes are sorted according to a space-filling curve.

The determinism guarantee of these systems is still quite fragile, because the insertion of a single instruction will produce a program that generates different outputs. Also, performance is sensitive to the task length. Devietti et al. show that system overheads can vary between 160%–250% depending on the task size parameter [17].

In contrast, systems like Grace and DThreads form their tasks based on synchronization instructions, which means that adding non-synchronization instructions will not change the decomposition of the program into tasks. However, this flexibility comes at a cost as tasks are now quite long, and load balancing becomes an issue. DThreads uses a sequential token passing algorithm to deterministically process synchronization events, so the entire sequence of instructions bounded by synchronization instructions is blocked waiting for the token. Kendo, which breaks tasks up into smaller pieces, can extract more parallelism by executing a prefix of instructions before the synchronization instruction. More recently, Cui et al. have proposed that users add performance hints akin to thread barriers to improve the load balancing of a deterministic scheduler [15].

Kendo, CoreDet, Determinator and some PBBS programs [7] have a tunable parameter that controls the task or round size, but no method to adaptively set that parameter based on observed execution. dOS uses instruction-based task formation, but it uses an adaptive algorithm like the one described in Section 3.2 to deterministically adjust the task size based on observed parallelism. Calvin uses a standard hardware two-bit predictor to dynamically increase task size when there is no synchronization in a task.

7. Conclusion

Deterministic execution of parallel programs has certain advantages such as reproducibility of results, which makes debugging easier. There is a substantial body of recent work on enforcing determinism at the programming language and programming model level, and at the system level through deterministic scheduling. We believe that any such system should provide three features: on-demand determinism, portability and parameter-freedom across platforms, which no existing system currently does.

In this paper, we considered the problem of ensuring deterministic execution for irregular programs, which are particularly challenging because task sizes are smaller and tasks communicate more frequently than in conventional parallel programs like the PARSEC benchmarks. Irregular programs have not been studied much in this context, and we showed that these programs do not scale when executed on a current determinism by scheduling system.

Our solution takes high-level non-deterministic programs written in the Galois model and implements determinism automatically by runtime scheduling. In many instances, the resulting programs have performance comparable to hand-written deterministic programs.

However, we showed on three different platforms that non-deterministic programs perform substantially better than their handwritten deterministic counterparts, demonstrating that there is a performance penalty for enforcing deterministic execution. This performance difference arises because the deterministic versions have less intra-task and inter-task locality. Therefore, we believe that determinism on demand, which leaves the choice between performance and determinism to the application user, is a reasonable design point.

References

[1] N. Amenta, S. Choi, and G. Rote. Incremental constructions con brio. In *Proc. Symp. on Computational Geometry (SCG)*, 2003.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proc. USENIX Conf. Operating Systems Design and Implementation*, OSDI, pages 1–16, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 53–64, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proc. USENIX Conf. Operating Systems Design and Implementation*, OSDI, pages 1–16, 2010.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *Proc. ACM SIGPLAN Conf. Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 81–96, 2009.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. Intl Conf. Parallel Architectures and Compilation Techniques*, PACT, pages 72–81, 2008.

[7] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP, pages 135–146, 2012.

[8] 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 *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP, 1995.

[9] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proc. USENIX Conf. Hot Topics in Parallelism*, HotPar, pages 4–4, 2009.

[10] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, POPL, pages 535–548, 2011.

[11] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proc. ACM SIGPLAN Conf. Object Oriented Programming Systems Languages and Applications*, OOPSLA, 2010.

[12] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4), Oct. 1984.

[13] B. V. Cherkassy and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. In *Proc. Intl Conf. Integer Programming and Combinatorial Optimization*, IPCO, pages 157–171, 1995.

[14] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4(5):287–421, 1989.

[15] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proc. ACM Symp. Operating Systems Principles*, SOSP, pages 388–405, 2013.

[16] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 85–96, 2009.

[17] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: a relaxed consistency deterministic computer. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[18] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[19] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. ACM SIGPLAN Conf. Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 388–402, 2003.

[20] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP, pages 207–216, 2008.

[21] D. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *IEEE Intl Symp. High Performance Computer Architecture*, HPCA, pages 333–334, 2011.

[22] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *IEEE Intl Symp. Performance Analysis of Systems and Software*, ISPASS, pages 65–76, 2009.

[23] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proc. ACM Symp. Parallelism in Algorithms and Architectures*, SPAA, pages 303–314, 2010.

[24] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Proc. ACM Symp. Operating Systems Principles*, SOSP, pages 327–336, New York, NY, USA, 2011.

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2009.

[26] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, pages 12–25, 2011.

[27] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. Intl Conf. Compiler Construction*, CC, pages 179–196, 2002.