Deterministic OpenMP for Race-Free Parallelism

Amittai Aviram and Bryan Ford
Yale University
{amittai.aviram, bryan.ford}@yale.edu

Recent deterministic parallel programming models show promise for their ability to replay computations and reproduce bugs, but they currently require the programmer to adopt restrictive or unfamiliar parallel constructs. Deterministic OpenMP (DOMP) is a new deterministic parallel environment built on the familiar OpenMP framework. By leveraging OpenMP's block-structured synchronization annotations, which are largely compatible with the constraints of a deterministic model, DOMP eases the parallelization of legacy serial code and preserves substantial compatibility with OpenMP software. A few OpenMP constructs, however, such as *atomic* and *critical*, are semantically nondeterministic and unsupported in DOMP. In three well-known parallel benchmark suites, we find that most (81%) uses of these nondeterministic constructs express programming *idioms* that are compatible with determinism but not directly expressible in OpenMP. DOMP therefore adds new OpenMP constructs to express such idioms deterministically, supporting pipelines and generalized reductions.

1 Introduction

Recent techniques to run parallel programs deterministically promise to manage race conditions and heisenbugs and to support exact replay of parallel software. A key desirable property is *race freedom*: execution timing or thread scheduling should not affect a program's behavior or results. New programming languages and type systems can guarantee race freedom [1, 2, 5, 9, 12, 13, 16, 21, 22, 24, 25], but they require programmers to rewrite legacy code and sometimes to adopt unfamiliar concepts. Deterministic schedulers [6, 7, 11] accommodate legacy code and make bugs reproducible, but they do not eliminate races from the programming model, as Section 2 explores.

For these reasons, we introduced a *working copies* programming model [3], which eliminates race conditions, while remaining compatible with existing languages such as C/C++. In this model, parallel threads work on isolated copies of data, then merge these copies at synchronization points, much as programmers use version control systems. Workspace consistency eliminates traditional read/write races, and transforms write/write races into *conflicts* that the runtime detects deterministically and reports as an error. Our first implementation of this model [4] offered no direct compatibility with existing parallel APIs, however, and the common pthreads API is a poor match for deterministic parallelism because it relies heavily on nondeterministic abstractions such as mutex locks and condition variables.

To address this compatibility challenge, we introduce Deterministic OpenMP (DOMP), a deterministic working copies environment based on the well-established OpenMP framework [20]. OpenMP is an attractive starting point because its fork-join parallelism and block structure fit in well with working-copies execution, and because most of OpenMP's core synchronization constructs, such as *parallel*, *loop*, and *sections*, are *naturally deterministic*. That is, programming logic alone determines which threads are involved and where the synchronization occurs in each thread's execution sequence.

DOMP does not support a few OpenMP constructs that are semantically nondeterministic, such as *atomic*, *critical*, and *flush*. In an analysis of the SPLASH, NPB, and PARSEC parallel benchmark suites, we find that nondeterministic constructs (37%) are less used than deterministic ones (63%), and that uses of these nondeterministic constructs rarely represent a semantic *need* for nondeterminism in the program. In those benchmarks written in OpenMP, *all* uses of nondeterministic constructs are to express programming idioms that are naturally deterministic in a higher-level sense.

For example, OpenMP's *reduction* clause is naturally deterministic, but only accepts scalar types and simple arithmetic or logical operators, so it cannot support operations over pointer variables. This limitation traditionally forces the OpenMP programmer to "escape" to lower-level nondeterministic synchronization constructs in order to build a more general reduction idiom. To remedy this lack while preserving the deterministic programming model, DOMP generalizes OpenMP's reductions to arbitrary operations and types.

DOMP's development is still early. One benchmark, matrix multiplication, shows a moderate 10% performance penalty on DOMP over standard OpenMP. On PARSEC's **blackscholes**, DOMP currently exhibits high overhead—but it also revealed a hitherto uncaught concurrency bug in the published code, illustrating the benefits of DOMP's conflict detection. Experience with Determinator [4] suggests that further optimizations could

enable DOMP to offer parallel determinism at an acceptable cost for a wider range of applications.

The contributions of this paper are: (a) the first deterministic parallel programming model fully compatible with existing languages such as C and at least partially compatible with an existing parallel API; (b) an analysis of the uses of synchronization abstractions in well-known parallel benchmark programs and their compatibility with deterministic programming models; and (c) a preliminary implementation and performance results suggesting that DOMP may point to a realistic and useful approach to deterministic parallel programming.

Section 2 traces work leading up to DOMP; Section 3 describes DOMP's semantics and extensions; Section 4 presents methods and results of our analysis of parallel benchmarks, suggesting that DOMP can make most nondeterministic abstractions unnecessary; Section 5 describes our prototype implementation and reports initial results; and Section 6 concludes.

2 Background and Related Work

A parallel program is *deterministic* if the input alone determines the output, regardless of extrinsic events such as the OS's thread scheduling. By itself, this definition allows for a range of behaviors, depending on the *synchronization* and *memory consistency models*.

Synchronization is deterministic if program logic determines how and at what points different threads interact, depending only on computation state and not on timing. A *fork* deterministically creates a child thread at a program-defined point in the parent's execution, for example. Similarly, a *join* deterministically combines the parent's and child's flows at program-defined points in *both* threads. Other common constructs such as mutex locks, condition variables, semaphores, monitors, and OpenMP's *atomic*, *critical*, and *flush*, are semantically nondeterministic: they allow a thread to signal or wake up an *unspecified*, nondeterministic recipient—e.g., the next holder of a lock—or to wait for an event from a nondeterministic source—e.g., any of several threads that might signal a condition variable.

Classic memory consistency models, including sequential consistency [18] and relaxed models [14, 23], also introduce nondeterminism by leaving memory access interleavings underspecified. That is, even if a program uses only deterministic synchronization abstractions (e.g., fork/join) and runs on sequentially consistent hardware, data races and execution timing can make the program exhibit any of an exponentially large variety of sequentially consistent memory access interleavings.

Deterministic schedulers such as DMP [11] and CoreDet [6] execute a semantically nondeterministic program repeatably, by artificially synthesizing *one particular* (arbitrary) interleaving of the program's synchronization and memory access events. Kendo [19] deterministically schedules synchronization but not memory accesses, yielding lower overheads but ensuring determinism only in programs free of memory access races.

Deterministic scheduling can reproduce races once detected, but it neither eliminates races nor guarantees that they *will* be detected. A program's behavior may still depend on the (deterministic) execution schedule in subtle ways not explicit in program logic, as in this example:

```
// Thread A:
{
  if (input_is_typical)
    do_a_lot();
  x++;
}
// Thread B:
{
  do_a_little();
  x++;
}
```

Under "typical" program inputs, abstracted here via '`input_is_typical`', a deterministic scheduler may always cause thread B to reach its increment of x while thread A is executing its long-running and non-conflicting `do_a_lot()`. But some particular "rare" input, which unit tests may not have covered, may cause the threads' increments to line up in the deterministic execution schedule, resulting in a classic data race and an "input-dependent heisenbug."

Grace [7] is a deterministic scheduler that emulates sequential consistency via speculative execution and transactional memory techniques. Determinator [4] and Revisions [10] avoid the complexity of speculative execution by straying from sequential consistency. These projects achieve acceptable overhead for some workloads, but constrain programs to a minimal set of deterministic synchronization primitives such as *fork/join* and *barrier*.

The SHIM language [12, 13, 25] implements a deterministic message passing model, avoiding the challenges of making shared memory deterministic, but also foregoing the programming convenience of the shared memory abstraction and requiring programmers to marshal data into explicit messages. Deterministic Parallel Java [9] offers shared memory, but requires the programmer to adopt a new Java type system, and to "prove" statically via typing rules that parallel code is race-free. Array Building Blocks [15] promise deterministic parallelism, but their proprietary nature hampers detailed inspection.

DOMP's approach has most in common with that of Determinator [4] and Revisions [10], operating in the workspace consistency model [3], while supporting a wider range of naturally deterministic synchronization abstractions to increase compatibility with legacy code.

3 DOMP Semantics

DOMP builds on OpenMP [20] to offer a parallel programming model with both an expressive API and race-free, naturally deterministic semantics. DOMP retains most OpenMP core constructs, but excludes OpenMP's few nondeterministic constructs. DOMP further extends OpenMP's API with deterministic constructs that can replace the most common uses of its excluded nondeterministic constructs.

3.1 Retained OpenMP Constructs

DOMP keeps OpenMP's *parallel*; work-sharing *loop*, *sections*, *barrier*; and combined *parallel* work-sharing directives. In both OpenMP and DOMP, the *parallel* directive and its combined work-sharing variants represent a fork-join pair enclosing a structured block, creating and then joining a *team* of concurrent threads. Under DOMP, however, between any two synchronization points, no two concurrent threads may write a new value to the same shared variable (whether directly or through a pointer); the execution runtime treats such a data race as an error. Moreover, each thread's writes to shared variables remains invisible to all concurrent threads until the next synchronization point—such as a *barrier* or the closing *join*. These rules guarantee the controlled flow of data from thread to thread.

The *master* directive is naturally deterministic, since it appoints a single thread, the "master" (parent of the other team threads), to execute the code, and since OpenMP's implied *barrier* at the end controls data transfer to the team. Since *single* allows the scheduler to appoint an arbitrary thread to execute the block, which may differ from run to run, DOMP makes *single* synonymous with *master*. Moreover, if the programmer disables the implicit closing *barrier* with a *nowait* clause, the master's changes remain invisible to the team until the next explicit synchronization point.

3.2 Excluded OpenMP Constructs

DOMP's semantics excludes OpenMP's *atomic*, *critical*, and *flush* constructs as naturally nondeterministic, since they imply that concurrent threads can have conflicting memory accesses. But how necessary are these abstractions? As we shall show in the next section, programmers often use them as low-level components of higher-level, deterministic idioms for which the parallel environment lacks suitable abstractions. We have identified two such idioms in particular, *reductions* and *pipelines*, representing 84% and 16%, respectively, of all invocations of these constructs in OpenMP code in our survey.

3.3 Reduction

Reductions are aggregate operations across threads; the result is only visible to the parent after the *join*, making data flow deterministic. OpenMP's *reduction* clause only allows scalar types and simple arithmetic or logical operators. DOMP has a generalized reduction:

$$\texttt{reduction}(\mathit{function} : \mathit{list})$$

The three arguments and return value of *function* have the same type as the variables in *list*. As in merge functions for revision types [10], the arguments to a DOMP reduction function correspond to a variable's state before the *fork*, after the *fork* in the master (parent) thread, and after the *fork* in a child thread to be joined with the master. The DOMP runtime evaluates a reduction at the *join*, in a reproducible order—a postorder binary tree walk—ensuring both a deterministic result and scalable performance without assuming the reduction function is associative or commutative.

To find the lexicographically first entry in a list of strings, we could write the following:

```
char *f(char *o, char *m, char *t)
{ char *tmp=(strcmp(m,t)<0)?m:t;
  return (strcmp(o,tmp)<0)?o:tmp); }
/*... */
first=strings[0];
#pragma omp parallel for
  reduction(f:strings)
{ for (int i=1;i<num_strings;i++)
    if (strcmp(strings[i],first)<0)
      first=strings[i]; }
```

The runtime invokes f at merge time, after which *first* points to the globally first string.

3.4 Pipeline

A pipeline is a sequence of repeated tasks, each dependent on the completion of a cycle of the task before it. With each task assigned to a different thread, data pass from thread to thread deterministically as each thread waits for input, processes it, and passes the output on. To express this, DOMP extends the *sections* construct with a *pipeline* clause and a required loop. The first section's thread starts immediately; each later thread runs its *section* each time the previous *section* has finished, until the pipeline is empty:

```
#pragma omp sections pipeline
{ while (more_work()) {
#pragma omp section
  { do_step_a();  }
#pragma omp section
  { do_step_b();  }
/* ... */
#pragma omp section
  { do_step_n();  } } }
```

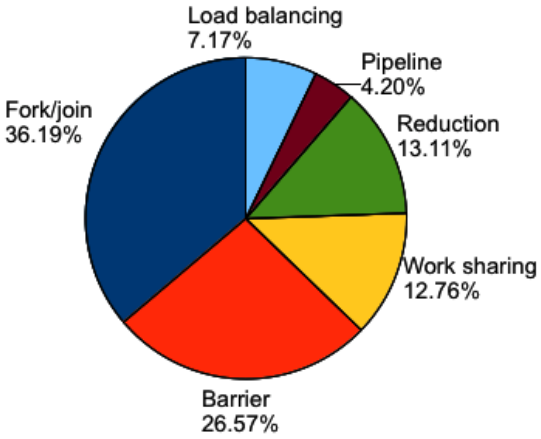

Figure 2: Types and uses of synchronization abstractions in SPLASH, NPB-OMP, and PARSEC programs.

4 Analysis of Parallel Benchmarks

How crucial are nondeterministic synchronization abstractions to the logic of parallel programs? To find out, we manually counted invocations of synchronization abstractions in the programs of three parallel benchmark suites—SPLASH [26], NPB-OMP [17], and PARSEC [8], choosing OpenMP versions of programs whenever possible and pthreads versions otherwise. We counted some matching pairs of events as single instances:

- A *fork* and its corresponding *join*
- A lock acquisition and its corresponding release
- A condition variable statement and associated lock acquisition

If a program duplicated a higher-level abstraction (e.g., *barrier*) with a lower-level fallback for compatibility, we counted only the higher-level abstraction.

We counted the locations where code invoked the naturally deterministic abstractions *fork/join* and *barrier* directly. Invocations of nondeterministic abstractions—locks and condition variables, as well as OpenMP *atomic*, *critical*, and *flush*—we grouped by the *idioms* in which they were used, for which we identified four classes (**D** = deterministic; **ND** = nondeterministic):

- *Work sharing* idioms (**D**), in pthreads code only since OpenMP offers work sharing constructs
- *Reduction* idioms (**D**), in pthreads code, or code violating the limitations of OpenMP reductions
- *Pipeline* idioms (**D**), as described in Section 3.4
- *Load balancing* idioms (**ND**), where an application schedules tasks to available threads in a pool.

Figures 1 and 2 summarize our findings. "Work-sharing" idioms occur in pthreads programs, which lack OpenMP's higher-level constructs; e.g., they lock a global integer and save its value as a thread ID for later task assignment, before incrementing and unlocking it. Likewise, pthreads code has to use locks to update variables to achieve the equivalent of reductions.

OpenMP's *reduction* constrains type and operation. Both pthreads and OpenMP lack a high-level construct to represent a pipeline, having instead to resort to spin loops and the nondeterministic *flush* construct (NPB **LU**) for synchronization. DOMP's extensions of OpenMP's *reduction* (3.3) and *sections* (3.4) will make it easier to avoid such nondeterministic workarounds.

Only user-level scheduling remains as a genuinely nondeterministic purpose for synchronization, representing just 19% of all nondeterministic synchronization invocations, or 7% of synchronization invocations overall, and none in OpenMP code. Possible extensions to DOMP to accommodate such needs are beyond our present scope.

5 Prototype Implementation

Our DOMP prototype is still in early development, thus far only supporting the *parallel* and *loop* constructs, without the *reduction* and *pipeline* extensions. This capability is sufficient to explore the feasibility of our design, based on the working-copies execution model, in a user-level library. In addition, DOMP already uncovered one concurrency bug in a well-known benchmark.

5.1 Design

DOMP adopts workspace consistency [3], pairing a deterministic synchronization model with a weak memory consistency model to achieve race freedom. In workspace consistency, each concurrent thread gets its own copy of shared state at its creation and operates on that copy in isolation. The runtime merges changes back into the parent's copy when the threads rejoin their parent. At join time, if two threads have independently modified the same memory location, DOMP defines this situation as a *conflict* and handles it as an application error, comparable to a divide by zero or illegal memory access.

We have modified GCC's OpenMP support library, libgomp, with the aim of creating a drop-in, deterministic substitute for OpenMP. Implementing DOMP this way introduces certain challenges. Since every thread must have its own private copy of shared variables, yet every variable must retain the same address for each thread, a lightweight threading model such as pthreads will not serve here as it does in libgomp. In a C-like language, a pointer variable may refer to an arbitrary location in memory, so the runtime must merge copies of the entire stack, heap, and static variable segments—excluding variables modified by the C library, as well as thread-private stacks and unmapped portions of any segment. Finally, applications may require a child thread to allocate heap storage for a shared pointer to which the

	SPLASH											NPB-OMP												PARSEC											
.	barnes	fmm	ocean	radiosity	volrend	water-nsquared	water-spatial	cholesky	fft	lu	radix	**BT**	**CG**	**DC**	**EP**	**FT**	**IS**	**LU**	**LU-HP**	**MG**	**SP**	**UA**	**blackscholes (omp)**	**bodytrack (omp)**	canneal	dedup	facesim	**ferret (omp)**	fluidanimate	freqmine	raytrace	streamcluster	swaptions	vips	x264
Fork/Join	1	1	1	3	5	1	1	1	1	1	1	10	14	2	2	8	3	9	15	11	14	59	1	4	1	4	2	7	13	7	1	1	1	1
Barrier	6	13	40	5	13	9	9	4	7	5	7						2	3							1						1	27		
Work sharing	6	28	2	5	8	4	6	7	1	1	1														1							3		
Reduction			2	5	6	4	4					1			1			2	2		2	46												
Pipeline																		10								10								4
Load balancing				14																							10			2			15	

Figure 1: Invocations of synchronization abstractions in source code of SPLASH, NPB, and PARSEC benchmark programs, including the naturally deterministic *fork/join* and *barrier* and nondeterministic primitives, further classified by the idioms in which they are used (*work sharing*, *reduction*, *pipeline*, *load balancing*).

parent has access (and which the parent can free) after the *join*, further complicating the task of checking for data races when merging children's working copies of the heap back into the parent's heap.

As in Grace [7], DOMP uses Unix *mmap* tricks to provide deterministic memory behavior with reasonable efficiency. DOMP implements a special *malloc* to manage the application's heap deterministically. Internally, a shared file holds heap space for a thread and its descendants. Children inherit valid heap pointer addresses from their parents, and a child can allocate its own heap for a parent's pointer, which remains valid after the runtime copies the child's heap back to the parent's at the join. For static variables, we linked labels into the application's object code to mark the start and end of linked segments relevant for conflict checking and merging.

5.2 Preliminary Results

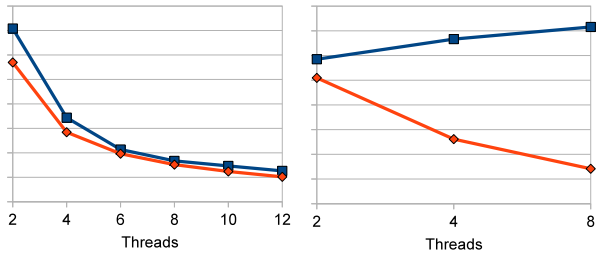

Figure 3: Performance comparison for DOMP (blue) and GCC's libgomp (orange) on matrix multiplication (left) and PARSEC's **blackscholes** (right).

We tested DOMP on simple parallel multiplication of 2048-by-2048 matrices written in ordinary OpenMP code and on the PARSEC **blackscholes** benchmark (OpenMP version), which compiled equally under standard and DOMP-enabled GCC. DOMP revealed a race condition on a shared variable assignment in the published **blackscholes** code, requiring modification. As Figure 3 shows, our current implementation adds about 10% to the running time over standard OpenMP for the matrix multiplication workload, but suffers from excess per-thread overhead on **blackscholes**. Performance results of comparable deterministic programming environments [4, 7, 10] make us optimistic that, with optimization, DOMP can achieve good performance.

6 Conclusion

The achievement of reliable, race-free parallel programs that burden the programmer neither with error-prone management of low-level sychronization nor with extensive code rewriting or refactoring requires a new deterministic parallel programming model, which Deterministic OpenMP introduces. Analysis of standard benchmarks suggests that DOMP's feature set can accommodate most parallel programming requirements while maintaining naturally deterministic semantics, and initial tests suggest that DOMP can be implemented efficiently.

Acknowledgments: This research was supported by the NSF under grant CNS-1017206.

References

[1] W.B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, Feb 1982.

[2] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: checking data sharing strategies for multithreaded C. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 149–158, New York, NY, USA, 2008. ACM.

[3] Amittai Aviram, Bryan Ford, and Yu Zhang. Workspace Consistency: A programming model for shared memory parallelism. In *2nd WoDet*, March 2011.

[4] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Determinator: OS support for efficient deterministic parallelism. In *9th OSDI*, October 2010.

[5] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA '00*, pages 382–400, 2000.

[6] Tom Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, March 2010.

[7] Emery D. Berger et al. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, October 2009.

[8] Christian Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *17th PACT*, October 2008.

[9] Robert L. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, October 2009.

[10] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA 2010*, pages 691–707, 2010.

[11] Joseph Devietti et al. DMP: Deterministic shared memory multiprocessing. In *14th ASPLOS*, March 2009.

[12] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *Transactions on VLSI Systems*, 14(8):854–867, August 2006.

[13] Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *DATE*, March 2008.

[14] Kourosh Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th ISCA*, pages 15–26, May 1990.

[15] Anwar Guloum et al. Array Building Blocks: A flexible parallel programming model for multicore and many-core architectures, September 2010.

[16] P. Hudak. Para-functional programming. *Computer*, 19(8):60–70, August 1986.

[17] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[18] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[19] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *14th ASPLOS*, March 2009.

[20] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[21] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[22] Paul Roe. *Parallel Programming using Functional Languages*. PhD thesis, University of Glasgow, feb 1991. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.7763&rep=rep1&type=pdf`.

[23] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 161–172, New York, NY, USA, 2007. ACM.

[24] Jacob T. Schwartz. The burroughs FMP machine, January 1980. Ultracomputer Note #5.

[25] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *EMSOFT*, pages 142–151, October 2006.

[26] Steven Cameron Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd ISCA*, pages 24–36, June 1995.