Internally Deterministic Parallel Algorithms Can Be Fast

Guy E. Blelloch* Jeremy T. Fineman† Phillip B. Gibbons‡ Julian Shun*

*Carnegie Mellon University †Georgetown University ‡Intel Labs, Pittsburgh

guyb@cs.cmu.edu jfineman@cs.georgetown.edu phillip.b.gibbons@intel.com jshun@cs.cmu.edu

Abstract

The virtues of deterministic parallelism have been argued for decades and many forms of deterministic parallelism have been described and analyzed. Here we are concerned with one of the strongest forms, requiring that for any input there is a *unique* dependence graph representing a trace of the computation annotated with every operation and value. This has been referred to as *internal determinism*, and implies a sequential semantics—*i.e.*, considering any sequential traversal of the dependence graph is sufficient for analyzing the correctness of the code. In addition to returning deterministic results, internal determinism has many advantages including ease of reasoning about the code, ease of verifying correctness, ease of debugging, ease of defining invariants, ease of defining good coverage for testing, and ease of formally, informally and experimentally reasoning about performance. On the other hand one needs to consider the possible downsides of determinism, which might include making algorithms (i) more complicated, unnatural or special purpose and/or (ii) slower or less scalable.

In this paper we study the effectiveness of this strong form of determinism through a broad set of benchmark problems. Our main contribution is to demonstrate that for this wide body of problems, there exist efficient internally deterministic algorithms, and moreover that these algorithms are natural to reason about and not complicated to code. We leverage an approach to determinism suggested by Steele (1990), which is to use nested parallelism with commutative operations. Our algorithms apply several diverse programming paradigms that fit within the model including (i) a strict functional style (no shared state among concurrent operations), (ii) an approach we refer to as *deterministic reservations*, and (iii) the use of commutative, linearizable operations on data structures. We describe algorithms for the benchmark problems that use these deterministic approaches and present performance results on a 32-core machine. Perhaps surprisingly, for all problems, our internally deterministic algorithms achieve good speedup and good performance even relative to prior nondeterministic solutions.

Categories and Subject Descriptors D.1 [*Concurrent Programming*]: Parallel programming

General Terms Algorithms, Experimentation, Performance

Keywords Parallel algorithms, deterministic parallelism, parallel programming, commutative operations, graph algorithms, geometry algorithms, sorting, string processing

PPoPP '12 February 25–29, 2012, New Orleans, Louisiana, USA.

1. Introduction

One of the key challenges of parallel programming is dealing with nondeterminism. For many computational problems, there is no inherent nondeterminism in the problem statement, and indeed a serial program would be deterministic—the nondeterminism arises solely due to the parallel program and/or due to the parallel machine and its runtime environment. The challenges of nondeterminism have been recognized and studied for decades [23, 24, 37, 42]. Steele's 1990 paper, for example, seeks "to prevent the behavior of the program from depending on any accidents of execution order that can arise from the indeterminacy" of asynchronous programs [42]. More recently, there has been a surge of advocacy for and research in determinism, seeking to remove sources of nondeterminism via specially-designed hardware mechanisms [19, 20, 28], runtime systems and compilers [3, 5, 36, 45], operating systems [4], and programming languages/frameworks [11].

While there seems to be a growing consensus that determinism is important, there is disagreement as to what degree of determinism is desired (worth paying for). Popular options include:

- *Data-race free* [2, 22], which eliminate a particularly problematic type of nondeterminism: the data race. Synchronization constructs such as locks or atomic transactions protect ordinary accesses to shared data, but nondeterminism among such constructs (*e.g.*, the order of lock acquires) can lead to considerable nondeterminism in the execution.
- *Determinate* (or *external determinism*), which requires that the program always produces the same output when run on the same input. Program executions for a given input may vary widely, as long as the program "converges" to the same output each time.
- *Internal determinism*, in which key aspects of intermediate steps of the program are also deterministic, as discussed in this paper.
- *Functional determinism*, where the absence of side-effects in purely functional languages make all components independent and safe to run in parallel.
- *Synchronous parallelism*, where parallelism proceeds in lock step (*e.g.*, SIMD-style) and each step has a deterministic outcome.

There are trade-offs among these options, with stronger forms of determinism often viewed as better for reasoning and debugging but worse for performance and perhaps programmability. Making the proper choice for an application requires understanding what the trade-offs are. In particular, is there a "sweet spot" for determinism, which provides a particularly useful combination of debuggability, performance, and programmability?

In this paper, we advocate a particular form of *internal determinism* as providing such a sweet spot for a class of nested-parallel (*i.e.*, nested fork-join) computations in which there is no

inherent nondeterminism in the problem statement. An execution of a nested-parallel program defines a dependence DAG (directed acyclic graph) that represents every operation executed by the computation (the nodes) along with the control dependencies among them (the edges). These dependencies represent ordering within sequential code sequences, dependencies from a fork operation to its children, and dependencies from the end of such children to the join point of the forking parent. We refer to this DAG when annotated with the operations performed at each node (including arguments and return values, if any) as the *trace*. Informally, a program/algorithm is *internally deterministic* if for any input there is a *unique* trace. This definition depends on the level of abstraction of the operations in the trace. At the most primitive level the operations could represent individual machine instructions, but more generally, and as used in this paper, it is any abstraction level at which the implementation is hidden from the programmer. We note that internal determinism does not imply a fixed schedule since any schedule that is consistent with the DAG is valid.

Internal determinism has many benefits. In addition to leading to external determinism [37] it implies a sequential semantics—*i.e.*, considering any sequential traversal of the dependence DAG is sufficient for analyzing the correctness of the code. This in turn leads to many advantages including ease of reasoning about the code, ease of verifying correctness, ease of debugging, ease of defining invariants, ease of defining good coverage for testing, and ease of formally, informally and experimentally reasoning about performance [3–5, 11, 19, 20, 28, 36, 45]. Two primary concerns for internal determinism, however, are that it may restrict programmers to a style that (i) is complicated to program, unnatural, or too special-purpose and (ii) leads to slower, less scalable programs than less restrictive forms of determinism. Indeed, prior work advocating less restrictive forms of determinism has cited these concerns, particularly the latter concern [25].

This paper seeks to address these two concerns via a study of a set of benchmark problems. The problems are selected to cover a reasonably broad set of applications including problems involving sorting, graphs, geometry, graphics and string processing. Our main contribution is to demonstrate that *for this wide body of problems, there exist fast and scalable internally deterministic algorithms, and moreover that these algorithms are natural to reason about and not complicated to code*.

Our approach for implementing internal determinism for these benchmarks is to use nested parallel programs in which concurrent operations on shared state are required to commute [42, 44] in their semantics and be linearizable [27] in their implementation. Many of the algorithms we implement use standard algorithmic techniques based on nested data parallelism where the only shared state across concurrent operations is read-only (*e.g.*, divide-and-conquer, map, reduce, and scan) [6]. However, a key aspect to several of our algorithms is the use of non-trivial commutative operations on shared state. The notion of commutativity has a long history, dating back at least to its use in analyzing when database transactions can safely overlap in time [44]. A seminal paper by Steele [42] discusses commutativity in the context of deterministic nested-parallel programs, showing that when applied to reads and writes on memory locations, commutativity of concurrent operations is sufficient to guarantee determinism.

Although there has been significant work on commutativity, there has been little work on the efficacy or efficiency of using non-trivial commutativity in the design of deterministic parallel algorithms. Much of the prior work on commutativity focuses on enforcing commutativity assuming the program was already written within the paradigm (*e.g.*, using type systems [12]), automatically parallelizing sequential programs based on the commutativity of operations [39, 40, 43], or using commutativity to relax the constraints in transactional systems [26, 30], an approach that does not guarantee determinism. In contrast, this paper identifies useful applications of non-trivial commutativity that can be used in the design of internally deterministic algorithms.

We describe, for example, an approach we refer to as *deterministic reservations* for parallelizing certain greedy algorithms. In the approach the user implements a loop with potential loop carried dependencies by splitting each iteration into *reserve* and *commit* phases. The loop is then processed in rounds in which each round takes a prefix of the unprocessed iterates applying the reserve phase in parallel and then the commit phase in parallel. Some iterates can fail during the commit due to conflicts with earlier iterates and need to be retried in the next round, but as long as the operations commute within the reserve and commit phases and the prefix size is selected deterministically, the computation is internally deterministic (the same iterates always fail).

We describe algorithms for the benchmark problems using these approaches and present performance results for our Cilk++ [31] implementations on a 32-core machine. Perhaps surprisingly, for all problems, our internally deterministic algorithms achieve good speedup and good performance even relative to prior nondeterministic and externally deterministic solutions, implying that the performance penalty of internal determinism is quite low. We achieve speedups of up to 31.6 on 32 cores with 2-way hyperthreading (for sorting). Almost all our speedups are above 16. Compared to what we believe are quite good sequential implementations we range from being slightly faster on one core (sorting) to about a factor of 2 slower (spanning forest). All of our algorithms are quite concise (20-500 lines of code), and we believe they are "natural" to reason about (understandable, not complicated, not special purpose). The paper presents code for two of the algorithms as illustrative examples; code for all of the algorithms (as well as complete descriptions of the benchmarks) can be found at `www.cs.cmu.edu/~pbbs`. We believe that this combination of performance and understandability provides significant evidence that internal determinism is a sweet spot for a broad range of computational problems.

The paper is organized as follows. Section 2 defines key terms and our programming model. Section 3 presents useful commutative building blocks. Section 4 describes the benchmark problems studied. Section 5 presents our approaches and algorithms. Our experimental study is in Section 6, and conclusions in Section 7.

2. Programming Model

This paper focuses on achieving *internally deterministic* behavior in "nested-parallel" programs through "commutative" and "linearizable" operations. Each of these terms limits the programs permitted by the programming model, but as Section 5 exhibits, the model remains expressive. This section defines each of these terms.

Nested parallelism. Nested-parallel computations achieve parallelism through the nested instantiation of fork-join constructs, such as parallel loops, parallel map, parbegin/parend, parallel regions, and spawn/sync. More formally, nested parallel computations can be defined inductively in terms of the composition of sequential and parallel components. At the base case a *strand* is a sequential computation. A *task* is then a sequential composition of strands and parallel blocks, where a *parallel block* is a parallel composition of tasks starting with a fork and ending with a join. Figure 1 shows an example of a nested-parallel program using a syntax similar to Dijkstra's parbegin [21].

A nested parallel computation can be modeled (a posteriori) as a series-parallel ***control-flow DAG*** over the operations of the computation: the tasks in a parallel block are composed in parallel, and the operations within a strand as well as the strands and parallel blocks of a task are composed in series in the order they are

```
1.  x := 0
2.  in parallel do
3.      {   r3 := AtomicAdd(x, 1)     }
4.      {   r4 := AtomicAdd(x, 10)
5.          in parallel do
6.              {   r6 := AtomicAdd(x, 100)    }
7.              {   r7 := AtomicAdd(x, 1000)   }
        }
8.  return x
```

Figure 1. A sample nested-parallel program. Here, the **in parallel** keyword means that the following two $\{\ldots\}$ blocks of code may execute in parallel. $\texttt{AtomicAdd}(x, v)$ atomically updates x to $x := x + v$ and returns the new value of x.

executed. We assume all operations take a state and return a value and a new state (any arguments are part of the operation). Nodes in the control-flow DAG are labeled by their associated operation (including arguments, but not return values or states). We say that an operation (node) u *precedes* v if there is a directed path from u to v in the DAG. If there is no directed path in either direction between u and v, then u and v are ***logically parallel***, meaning that they *may* be executed in parallel.

The support of nested parallelism dates back at least to Dijkstra's parbegin-parend construct. Many parallel languages support nested parallelism including NESL, Cilk, the Java fork-join framework, OpenMP, the TBB, and TPL. Although not appropriate for certain types of parallelism, *e.g.*, pipeline parallelism, nested parallelism has many theoretical and practical advantages over more unstructured forms of parallelism, including simple schedulers for dynamically allocating tasks to cores, compositional analysis of work and span, and good space and cache behavior (*e.g.*, [1, 6, 8, 10]).

Languages with nested parallelism rely on runtime schedulers to assign subcomputations to cores. Whereas these runtime schedulers are inherently nondeterministic to handle load balancing and changes in available resources, our goal is to guarantee that the program nevertheless behaves deterministically.

Internal determinism. We adopt a strong notion of determinism here, often called internal determinism [35]. Not only must the output of the program be deterministic, but all intermediate values returned from operations must also be deterministic. We note that this does not preclude the use of pseudorandom numbers, where one can use, for example, the approach of Leiserson *et al.* [33] to generate deterministic pseudorandom numbers in parallel from a single seed, which can be part of the input.

This paper defines determinism with respect to abstract operations and abstract state, not with respect to machine instructions and memory state. Nevertheless, the definition supplied here is general and applies to both cases. The difference hinges on the notion of "equivalence." Given a definition of equivalent operations, states, and values, we define internal determinism as follows.

For a (completed) computation its ***trace*** is the final state along with the control-flow DAG on which operation nodes are (further) annotated with the values returned (if any). Figure 2 shows two traces corresponding to executions of the program shown in Figure 1. Two control-flow DAGs are equivalent if they have the same graph structure and corresponding nodes are labeled with equivalent operations. Two traces are ***equivalent traces*** if they have equivalent final states, equivalent control-flow DAGs, and corresponding DAG nodes are annotated with equivalent return values.

Definition 1. *A program is **internally deterministic** if for any fixed input I, all possible executions with input I result in equivalent traces.*

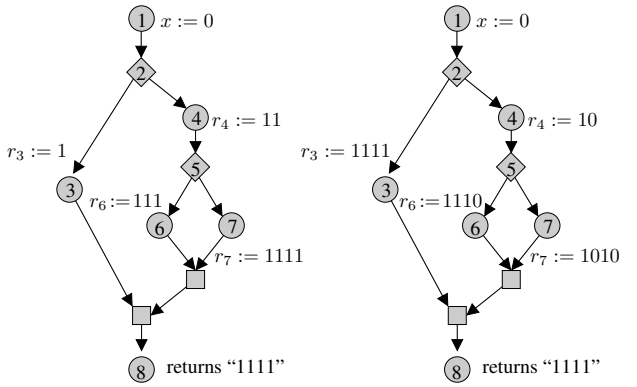


Figure 2. Two possible traces for the program in Figure 1. The diamonds, squares, and circles denote forks, joins, and data operations, respectively. Nodes are numbered by line number, as a short hand for operations such as $\texttt{AtomicAdd}(x, 1)$. The left trace corresponds to the interleaving/schedule $1, 2, 3, 4, 5, 6, 7, 8$, whereas the right trace corresponds to $1, 2, 4, 5, 7, 6, 3, 8$. Because the intermediate return values differ, the program is not internally deterministic. It is, however, externally deterministic as the output is always the same. If `AtomicAdd` did not return a value, however, then the program would be internally deterministic.

Note that since the parallelism is dynamic, a nondeterministic program may result in dramatically different DAGs. Because all decisions in a computation are based only on the result of operations performed, however, if operations return equivalent results despite different schedulings, then the structure of the DAG is guaranteed to remain the same.

For primitive types like integers, it is clear what equivalence means. When working with objects and dynamic memory allocation, however, a formal definition of equivalent objects and states becomes more complicated, and not within the scope of this paper. Informally, when we say that states or values are equivalent, we mean semantically equivalent, *i.e.*, that no sequence of valid operations can distinguish between them (see, *e.g.*, [26]).

Commutativity. Internally deterministic programs are a subset of parallel programs, and thus programming methodologies that yield internal determinism restrict a program's behaviors. The methodology we adopt in this paper is to require all logically parallel accesses of shared objects to use operations that commute. The fact that this restriction yields internally deterministic programs is observed in many works, see for example [15, 40, 42] among others.

We adopt Steele's notation and definition of commutativity [42]. We use $f(S) \to S' \Rightarrow v$ to denote that when the operation f is executed (without any concurrent operations) starting from system (object) state S, the system transitions to state S' and f returns the value v. To simplify notation, operations not returning values are viewed as returning $v = \emptyset$.

Definition 2. *Two operations f and g **commute with respect to state** S if the order in which they are performed does not matter. That is, if*

$$f(S) \to S_f \Rightarrow v_f$$
$$g(S_f) \to S_{fg} \Rightarrow v_g$$

and

$$g(S) \to S'_g \Rightarrow v'_g$$
$$f(S'_g) \to S'_{gf} \Rightarrow v'_f$$

then f and g commute with respect to S if and only if $S_{fg} = S'_{gf}$, $v_f = v'_f$, and $v_g = v'_g$, where "=" here denotes equivalence. (Note that there is no requirement that $S_f = S'_g$.)

Moreover, we say that two operations ***commute*** if they commute with respect to *all* valid states S. It is possible to relax this definition (*e.g.*, [26, 44]), but we found this definition sufficient for our purposes.

Linearizability. Commutativity is not a sufficient condition for deterministic behavior, as commutativity alone does not guarantee that the implementation of the operations work correctly when their instructions are interleaved in time. To guarantee safety of concurrent execution of operations we use the standard definition of linearizability [27], which enforces atomicity of the operations. In our setting, operations are concurrent if and only if they are logically parallel. Thus, linearizability guarantees that there is a total order (or *history*), H, of the annotated operations in a trace T such that H is a legal sequential execution of those operations, starting from the initial state. That is, (i) H is a valid scheduling of T's control-flow DAG, and (ii) each annotated operation in T remains legal (including its return value) when executed atomically in the order of H. We note that linearizability is a property of the implementation and not the semantics of the operation (*e.g.*, two insertions into a dictionary might semantically commute, but an implementation might fail when interleaved). One way to guarantee linearizability is to use a lock around all commuting operations, but this is inefficient. In this paper we use only non-blocking techniques to achieve linearizability among commuting operations. We however do not guarantee that all commuting operations are linearizable, just that the logically parallel ones are.

Summary. The model we will use for internally deterministic behavior is summarized by the following theorem.

Theorem 1. *Let P be a nested-parallel program. If for all inputs, all logically parallel operations commute and are linearizable, then P is internally deterministic.*

Proof. (sketch) Consider any fixed input I and any fixed (completed) execution of P with input I. Let G (T) be the resulting control-flow DAG (trace, respectively), and let H be its linearizability history. We will show that T is equivalent to a canonical trace T^* obtained by executing P with input I using only a single core. Let G^* and H^* be the control-flow DAG and linearizability history for T^*. We show by induction on the length of H^* that (i) G and G^* are equivalent and (ii) H permuted to match the order in H^* of equivalent nodes is also a linearizability history for T, implying equivalent return values. We construct such a permutation, H', inductively, with $H' = H$ initially. Assume inductively that (i) the subgraph of G^* corresponding to the nodes in $H^*[1..i]$ has an equivalent subgraph in G, and (ii) H' is a linearizability history for T such that $H'[1..i]$ and $H^*[1..i]$ are equivalent ($[j..k]$ denotes subsequence). Consider $i+1$, and let σ^* be the $i+1$'st annotated node in H^*. It follows inductively that there is a node σ in T with equivalent parent(s) and an equivalent operation, say the jth node in H'. If $j = i+1$, we are done, so assume $j > i+1$. None of the nodes in $H'[i+1..j-1]$ can precede or be preceded by σ, so σ must commute with each such node. Thus, σ can be pairwise swapped up to position $i+1$ in H' while preserving a linearizability history, establishing both inductive invariants. The argument is readily extended to show the equivalence of the final states by augmenting each execution with operations that read the final state. The theorem follows. □

Our approach is similar to previous models for enforcing deterministic behavior [15, 42] except that in Steele [42] commutativity is defined in terms of memory operations and memory state, and in Cheng *et al.* [15] commutativity is defined with respect to critical sections and memory state. Here we define commutativity in terms of linearizable abstract operations and abstract state.

3. Commutative Building Blocks

Achieving deterministic programs through commutativity requires some level of (object or operation) abstraction. Relying solely on memory operations is doomed to fail for general purpose programming. For example requiring a fixed memory location for objects allocated in the heap would severely complicate programs and/or inhibit parallelism, possibly requiring all data to be preallocated. Instead, this section defines some useful higher-level operations that we use as commutative operations in many of our algorithms. They are all defined over abstract data types supporting a fixed set of operations. We also describe non-blocking linearizable implementations of each operation. These implementations do not commute at the level of single memory instructions and hence the abstraction is important.

Priority write. Our most basic data type is a memory cell that holds a value and supports a priority write and a read. The priority write on a cell x, denoted by $x.\texttt{pwrite}(v)$ updates x to be the maximum of the old value of x and a new value v. It does not return any value. $x.\texttt{read}()$ is just a standard read of the cell x returning its value. We often use priority write to select a deterministic winner among parallel choices, *e.g.*, claiming a next-step neighbor in breadth first search (Section 5.3).

Any two priority writes $x.\texttt{pwrite}(v_1)$ and $x.\texttt{pwrite}(v_2)$ commute, in accordance with Definition 2, because (i) there are no return values, and (ii) the final value of x is the maximum among its original value, v_1, and v_2, regardless of which order these operations execute. A priority write and a read do not commute since the priority write can change the value at the location. We implement non-blocking and linearizable priority writes using a compare and swap. With this implementation the machine primitives themselves do not commute.

Priority reserve. In our "deterministic reservation" approach described later in Section 5, multiple program loop iterates attempt to reserve the same object in parallel, and later the winner operates on the reserved object. For deterministic reservations we use a data type that supports three operations, a priority reserve ($x.\texttt{reserve}(p)$), a check ($x.\texttt{check}(p)$), and a check-and-release ($x.\texttt{checkR}(p)$), where p is a priority. As with a priority write, a higher priority value overwrites a lower priority and hence the highest priority will "reserve" the location. The one difference is that we require a unique priority tag $\perp$ to denote when the location is currently unreserved. The priority $\perp$ has the lowest priority, and it is invalid to make a `pwrite` call with $p = \perp$. As with `pwrite`, any number of `reserve`s commute, and we implement a linearizable non-blocking version using compare and swap.

The $x.\texttt{checkR}(p)$ call requires $p \neq \perp$. If the current value at location x has priority p, then the reservation is released (*i.e.*, the value $\perp$ is written to x), and TRUE is returned to indicate that p was the highest priority reservation on x. If the current priority is not p, then the state does not change and FALSE is returned. Operations $x.\texttt{checkR}(p_1)$ and $x.\texttt{checkR}(p_2)$ commute if and only if $p_1 \neq p_2$. A `check` is the same as a `checkR` without the release and commutes in the same way. A priority reserve and either form of check do not commute.

Our algorithms ensure that for any given location, (i) priority reserves are not called logically in parallel with either form of check, and (ii) all logically parallel operations use distinct priorities. Thus, the commutativity and resulting internal determinism extend to those algorithms.

Dynamic map. The purpose of our dynamic map is to incrementally insert keyed elements and, once finished inserting, to return an array containing a pseudorandom permutation of these elements, omitting duplicates. A dynamic map supports two operations: $M.\texttt{insert}(x)$, which inserts keyed element x into the map M without returning any value, and $M.\texttt{elements}()$, which returns an arbitrary, but deterministic, permutation of all the elements in the map M. The map removes duplicate keys on insert: if elements y and x have the same key and y is already in the map when $M.\texttt{insert}(x)$ is called, one of the elements (chosen deterministically based on a user specified priority) is discarded.

We implement our dynamic map using a history-independent hash table [7]. In a history-independent data structure the final layout does not depend on the operation order. In particular, the key of each element is treated as a priority, and the hash table is equivalent to one in which all insertions were performed sequentially in a nonincreasing priority order using linear probing. Elements are inserted by first hashing the key and going to the corresponding hash location, then scanning consecutive hash-table slots until finding either an empty slot or a slot containing and equal- or lower-priority element. If empty, the new element is inserted and the operation completes. If the slot is occupied by an equal-priority element, either the new or old element is discarded (deterministically based on priority) and the operation completes. If the slot is occupied by a lower-priority element, the higher-priority element is put in that slot (using compare and swaps to provide linearizability), and the lower-priority element is evicted. The linear probe continues to find a slot for the lower-priority element. An `elements` call simply filters the underlying array (using the parallel filter operation discussed in Section 5), finding all the nonempty slots and placing them in order in a return array. Our implementation is non-blocking requiring no locks.

To see that two `insert`s commute, it is easy to show inductively that after each `insert`, the hash table is identical to one in which those elements present were inserted in priority order. This property implies that the ordering between two insertions does not matter. The $M.\texttt{insert}(x)$ operation does not commute with $M.\texttt{elements}()$ operation since for some states of S, x is not in M and will affect the result of `elements`.

Disjoint sets. Our spanning-forest algorithms rely on a structure for maintaining a collection of disjoint sets corresponding to connected components. Each set is associated with a unique element acting as the identifier for the set. A disjoint-set data type supports two operations: a find and a link. For an instance F, the $F.\texttt{find}(x)$ operation returns the set identifier for the set containing x. The $F.\texttt{link}(S, x)$ operation requires that S be a set identifier and the set containing x be disjoint from the set S. It logically unions the set S with the set containing x such that the identifier for the resulting unioned set is the identifier of the set containing x. Here, x and S denote references or pointers to elements in the sets.

We implement an instance F of the disjoint set data type as a collection of trees with parent pointers, where the root of each tree acts as a unique identifier for the set [17]. A $F.\texttt{find}(x)$ operation simply follows parent pointers up the tree and returns the root. It may also perform path compression [17], which points nodes along the query-to-root path directly to the root, thereby accelerating future queries. A $\texttt{link}(S, x)$ operation is implemented by pointing S to the root-node of the set containing x.

Two `find` operations commute with each other as they cause no semantic modifications—i.e. any changes to the pointer structure caused by path compression cannot be discerned by future operations on F. Two `link` operations commute with each other as long as they do not share the same first argument. That is to say, $F.\texttt{link}(S_1, x_1)$ and $F.\texttt{link}(S_2, x_2)$ commute as long as $S_1 \neq S_2$; having x_1 and x_2 be equal or from the same set is allowed, as is having x_1 in set S_2 or x_2 in set S_1. The $\texttt{link}(S_1, x_1)$ and $\texttt{find}(x_2)$ only commute if $x_1 = x_2$.

We now consider linearizability. Even with path compression, `find` operations are linearizable (and non-blocking) since there is only one possible update to each pointer (the *a priori* root of the tree). This requires no compare and swap or any other special memory operations. Logically parallel `link` operations with distinct first arguments, and no cycles among the linked sets, are also linearizable and non-blocking with no special memory operations since they only require updating a pointer which is not shared by any other logically parallel operation. In our implementation we do not guarantee that `find`s and `link`s are linearizable. Hence, in our algorithms that use disjoint sets, `find`s are never logically parallel with `link`s: they alternate phases of only `find`s and only `link`s.

We note that we use an asymmetric `link` operation instead of the standard symmetric `union`. This is because `union` does not commute in our definition which requires two operations to commute for all start states. In a more relaxed definition of commutativity, `union` can be made to commute [30].

4. Benchmark Problems

For testing the utility of nested-parallel internally deterministic algorithms we use a set of *problem-based* benchmarks. These benchmarks are defined in terms of the problem they solve instead of any particular code or algorithm to solve the problem. We feel that this is important for our purposes since it might be that very different algorithmic approaches are suited for a deterministic algorithm vs. a nondeterministic algorithm. The benchmark suite is selected to cover a reasonable collection of fundamental problems. The focus, however, is on problems involving unstructured data since there is already very good coverage for such benchmarks for linear algebra and typically deterministic algorithms are much simpler for these problems. The problems are selected to be simple enough to allow reasonably concise implementations, but interesting enough to be non-trivial. For all problems we use a variety of different inputs and avoid just random inputs. Here we define the problems.

Comparison Sort: For a sequence S and comparison function $<$ defining a total order on elements of S, return the values of S sorted by $<$. Sorting is a fundamental problem and a subroutine in many algorithms. The benchmark code must work with any element type and comparison function.

Remove Duplicates: For a sequence of elements of type t, a hash function $h : t \rightarrow int$, and comparison function f, return a sequence in which any duplicates (equal valued elements) are removed. This is an example of a dictionary-style operation that can use hashing.

Breadth First Search: For a connected undirected graph G, and source vertex s, return a breadth-first-search (BFS) tree, rooted at s, of the vertices in G.

Spanning Forest: For an undirected graph $G = (V, E)$, return edges $F \subset E$, such that for each connected component $C_i = (V_i, E_i)$ in G, a spanning tree T_i ($|T_i| = |V_i|-1$) of C_i is contained in F. Furthermore, $|F| = \sum_{C_i \subset G}(|V_i| - 1)$.

Minimum Spanning Forest: For an undirected graph $G = (V, E)$ with weights $w : E \rightarrow \Re$, return a spanning forest of minimum total weight.

Maximal Independent Set: For a connected undirected graph $G = (V, E)$, return $U \subset V$ such that no vertices in U are neighbors and all vertices in $V \setminus U$ have a neighbor in U. This is an important subroutine in many parallel algorithms because it can be used to identify a set of vertices that can be operated on deterministically in parallel (due to disjoint edge sets).

Triangle Ray Intersect: For a set of triangles T and rays R in three dimensions, return the first triangle each ray intersects, if any.

Problem	D&C	Reduce	Scan	Filter	DR	CL
Comparison Sort	yes		yes			
Remove Duplicates				yes		DM
Breadth First Search			yes	yes		PW
Spanning Forest				yes	yes	DS
Min Spanning Forest	sub			yes	yes	DS
Max Independent Set		yes		yes	yes	
Triangle Ray Intersect	yes		yes	yes		
Suffix Array	sub	yes	yes	yes		
Delaunay Triangulation	sub	yes	sub	yes	yes	
Delaunay Refine		yes		yes	yes	DM
N-body	yes	yes	yes			
K-Nearest Neighbors	sub			yes		

Table 1. Techniques used in our algorithms for each of the benchmarks. D&C indicates divide-and-conquer; Reduce, Scan and Filter are standard collection operations; DR indicates deterministic reservations; and CL indicates the use of a non-trivial commutative and linearizable operation other than reservations: dynamic map (DM), disjoint sets (DS), or priority write (PW). **sub** indicates that it is not used directly, but inside a subroutine, *e.g.*, inside a sort. See Section 5 for further details.

This is a common operation in graphics and is the most widely used special case of ray casting.

Suffix Array: For a string S of n characters return an equal length integer array A that specifies the sorted order of the suffixes of S. This is an important operation used in many applications in computational biology, compression, and string processing.

Delaunay Triangulation: For a set of n points in two dimensions, return a triangulation such that no point is contained in the circumcircle of any triangle in the triangulation [18]. Delaunay triangulations are likely the most widely used partitioning of space in two and three dimensions and used in many CAD applications.

Delaunay Refine: For a Delaunay Triangulation on a set of n points, and an angle α, add new points such that in the resulting Delaunay Triangulation, no triangle has an angle less than α.

N-body: For a set of n point sources in three dimensions, each point p with coordinate vector $\vec{p}$ and a mass m_p, return the force induced on each one by the others based on the Coulomb force $\vec{F_p} = \sum_{q \in P, q \neq p} m_q m_p (\vec{q} - \vec{p})/||\vec{q} - \vec{p}||^3$. The N-body problem is important in protein folding, astrophysics, and slight generalizations are now often used for solving PDEs.

K-Nearest Neighbors: For n points in two or three dimensions, and a parameter k, return for each point its k nearest neighbors (euclidean distance) among all the other points. The problem is fundamental in data analysis and computational geometry.

5. Internally Deterministic Parallel Algorithms

In this section we describe the approaches we used when designing our internally deterministic parallel algorithms and outline the resulting algorithms for each of the benchmarks. Many of the approaches used are standard, but we introduce what we believe to be a new approach for greedy algorithms based on deterministic reservations. This approach plays a key role in our implementation of five of the problems. We also make use of our commuting and linearizable implementations of various operations for five problems. Table 1 summarizes what approaches/techniques are used in which of our algorithms.

5.1 Nested Data Parallelism and Collection Operations

The most common technique throughout the benchmark implementations is the use of nested data parallelism. This technique is applied in a reasonably standard way, particularly in the use of fork-join and parallel loops (with arbitrary nesting) in conjunction with parallel operations on collections. For the operations on collections we developed our own library of operations on sequences. We make heavy use of divide and conquer. In the divide-and-conquer algorithms we almost always use parallelism within the divide step (to partition the input data), and/or the merge step (to join the results), typically using the collection operations in our sequence library.

The three collection operations `reduce`, `scan`, and `filter` are used throughout our algorithms. As is standard, `reduce` takes a sequence S and a binary associative function f and returns the "sum" of elements with respect to f, while `scan` (prefix sum) takes a sequence S and a function f and returns a sequence of equal length with each element containing the sum with respect to f of all preceding elements in S. Our implementations of `reduce` and `scan` are deterministic even if f is not associative—*e.g.*, with floating point addition. The `filter` operation takes a sequence S and a function f returning a boolean and returns a new sequence containing only the elements e for which $f(e)$ is true, in the same order as in S. Filter uses a scan in its implementation.

Reduce is used to calculate various "sums": *e.g.*, to calculate the bounding box (maximum and minimum in each coordinate) of a set of points. Filter is used in most of our algorithms. In the divide-and-conquer algorithms it is typically used to divide the input into parts based on some condition. In the other algorithms it is used to filter out elements that have completed or do not need to be considered. It plays a key role in deterministic reservations. Scan is used in a variety of ways. In the sorting algorithm it is used to determine offsets for the sample sort buckets, in the suffix array algorithm it is used to give distinct elements unique labels, and in the breadth first search algorithm it is used to determine the positions in the output array to place distinct neighbor arrays.

5.2 Deterministic Reservations

Several of our algorithms (maximal independent set, spanning forest, minimum spanning forest, Delaunay triangulation, and Delaunay refine) are based on a greedy sequential algorithm that processes elements (*e.g.*, vertices) in linear order. These can be implemented using speculative execution on a sequential loop that iterates over the elements in the greedy order.

Various studies have suggested both compiler [39, 40] and runtime techniques [25, 43] to automate the process of simulating in parallel the sequential execution of such a loop. These approaches rely on recognizing at compile and/or run time when operations in the loop iterates commute and allowing parallel execution when they do. Often the programmer can specify what operations commute. We are reasonably sure that the compiler-only techniques would not work for our benchmark problems because the conflicts are highly data dependent and any conservative estimates allowing for all possible conflicts would serialize the loop. The runtime techniques typically rely on approaches similar to software transactional memory: the implementation executes the iterations in parallel or out of order but only commits any updates after determining that there are no conflicts with earlier iterations. As with software transactions, the software approach is expensive, especially if required to maintain strict sequential order. In fact in practice the suggested approaches typically relax the total order constraint by requiring only a partial order [39], potentially leading to nondeterminism. A second problem with the software approach is that it makes it very hard for the algorithm designer to analyze efficiency—it is possible that subtle differences in the under-the-hood conflict resolution could radically change which iterates can run in parallel.

We present an approach, called *deterministic reservations*, that gives more control to the algorithm designer and fits strictly within

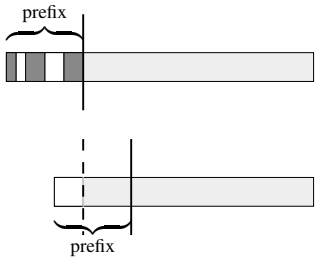


Figure 3. A generic example of deterministic reservations. Top and bottom depict the array of iterates during consecutive rounds. In each round, a prefix of some specified size is selected. All of these prefix iterates perform the reserve component. Then they all perform the commit component. The dark regions in the top array represent iterates that successfully commit. All uncommitted iterates (shown in white) are packed towards the right, as shown in the bottom array. The next round then begins by selecting a prefix of the same size on the bottom array.

the nested-parallel framework (needing neither special compiler nor runtime support). In this approach the algorithm designer controls exactly on what data the conflicts occur and these conflicts are deterministic for a given input. The generic greedy algorithm for deterministic reservations works as follows, illustrated in Figure 3. It is given a sequence of iterates (*e.g.*, the integers from 0 to $n-1$) and proceeds in rounds until no iterates remain. Each round takes any prefix of the remaining unprocessed iterates, and consists of two phases that are each parallel loops over the prefix, followed by some bookkeeping to update the sequence of remaining iterates. The first phase executes a *reserve* component on each iterate, using a priority reserve (`reserve`) with the iterate priority, in order to reserve access to data that might interfere (involve non-commuting or non-linearizable operations) with other iterates. The second phase executes a *commit* component on each iterate, using a `check` to see if the reservations succeeded, and if the required reservations succeed then the iterate is processed, otherwise it is not. Typically updates to shared state (at the abstraction level available to the programmer) are only made if successful. After running the commit phase, the processed iterates are removed. In our implementation the unprocessed iterates are kept in a contiguous array ordered by their priority. Selecting a prefix can therefore just use a prefix of the array, and removing processed iterates can be implemented with a `filter` over the boolean results of the second phase.

The specifics of the reserve and commit components depend on the application. The work done by the iterate can be split across the two components. We have found, however, that in the unstructured problems in the benchmarks just determining what data might interfere involves most of the work. Therefore the majority of the work ends up in the reserve component. In most cases all reservations are required to succeed, but we have encountered cases in which only a subset need to succeed (*e.g.*, our minimum spanning-forest code reserves both endpoints of an edge but only requires that one succeeds).

We note that the generic approach can select any prefix size including a single iterate or all the iterates. There is a trade off, however between the two extremes. If too many iterates are selected for the prefix, then many iterates can fail. This not only requires repeated effort for processing those iterates, but can also cause high-contention on the reservation slots. On the other hand if too few iterates are selected then there might be insufficient parallelism. Clearly the amount of contention depends on the specific algorithms and likely also on the input data.

As long as the prefix size is selected deterministically and all operations commute and are linearizable within the reserve phase and separately within the commit phase, a program will be internally deterministic. This means the algorithm designer only needs to analyze commutativity/linearizability within each phase. In our code we have implemented a function `speculative_for` that takes four arguments: a structure that implements the `reserve` and `commit` components (both taking an index as an argument), a start index, an end index, and a prefix size.

5.3 Algorithms

We now describe each of the algorithms we use to implement the benchmarks discussed in Section 4. In all cases we considered a variety of algorithms and selected the one we felt would perform the best. In many cases we arrived at the algorithm discussed after trying different algorithms. In all cases the algorithms are either motivated by or directly use results of many years of research on parallel algorithm design by many researchers. Due to limitations of space we only very briefly describe any algorithms that mostly use previous ideas.

Comparison Sort: We use a low-depth cache-efficient sample sort [9]. The algorithm (1) partitions the input into $\sqrt{n}$ blocks, (2) recursively sorts each block, (3) selects a global sample of size $\sqrt{n}\log n$ by sampling across the blocks, (4) sorts the sample, (5) buckets each of the blocks based on the sample, (6) transposes the keys so keys from different blocks going to the same bucket are adjacent, and (7) recursively sorts within the buckets. The transpose uses a cache-efficient block-transpose routine. When the input is small enough, quicksort is used. The algorithm is purely nested parallel. There is nesting of the parallelism (divide-and-conquer) in the overall structure, in the merge used for bucketing blocks, in the transpose, and in the quicksort.

Remove Duplicates: We use a parallel loop to concurrently `insert` the elements into the dynamic map described in section 3. This data structure already removes all duplicates internally and returns the distinct elements with a call to `elements` (which internally uses a `filter`). The ordering returned by the routine is deterministic but does not correspond to the input ordering in any natural way and different hash functions will give different orderings. We set the hash table size to be twice the size of the input.

Breadth First Search: We use a level-ordered traversal of the graph that guarantees the same BFS tree as the standard sequential queue-based algorithm [17]. In level-order traversal each vertex u adds each of its unvisited neighbors v to the next frontier and makes u the parent of v in the BFS tree. In standard parallel implementations of BFS [32, 39] each level is processed in parallel and nondeterminism arises because vertices at one level might share a vertex v at the next level. These vertices will attempt to add v to the next frontier concurrently. By using a compare-and-swap or similar operation, it is easy to ensure that a vertex is only added once. However, which vertex adds v depends on the schedule, resulting in internal nondeterminism in the BFS code and external nondeterminism in the resulting BFS tree.

We avoid this problem by using a priority write. The vertices in the frontier are prioritized by their position in the array and we process each level in two rounds. In the first round each vertex in the frontier writes its priority to all neighbors that have not been visited in previous rounds. In the second round each vertex v in the frontier reads from each neighbor u the priority. If the priority of u is v (v is the highest priority neighbor in the frontier), then we make v the parent of u and add u to the next frontier. The neighbors are added to the next frontier in the priority order of the current frontier. This uses a `scan` to open enough space for each neighbor list, and maintains the same ordering on every frontier as the sequential queue-based algorithm maintains.

```
struct STStep {
  int u;  int v;
  edge *E;  res *R;  disjointSet F;
  STStep(edge* _E, disjointSet _F, res* _R)
    : E(_E), R(_R), F(_F) {}

  bool reserve(int i) {
    u = F.find(E[i].u);
    v = F.find(E[i].v);
    if (u == v) return 0;
    if (u > v) swap(u,v);
    R[v].reserve(i);
    return 1;}

  bool commit(int i) {
    if (R[v].check(i)) { F.link(v, u); return 1;}
    else return 0;  }
};

void ST(res* R, edge* E, int m, int n, int psize) {
  disjointSet F(n);
  speculative_for(STStep(E, F, R), 0, m, psize);
}
```

Figure 4. C++ code for spanning forest using deterministic reservations (with its operations `reserve`, `check`, and `speculative_for`), where $\mathtt{m} = |E|$ and $\mathtt{n} = |V|$.

Spanning Forest: Sequentially a spanning forest can be generated by greedily processing the edges in an arbitrary order using a disjoint sets data structure. When an edge is processed if the two endpoints are in the same component (which can be checked with `find`) it is removed, otherwise the edge is added to the spanning forest and the components are joined (with `union`). This algorithm can be run in parallel using deterministic reservations prioritized by the edge ordering and will return the exact same spanning forest as the sequential algorithm. The idea is simply to reserve both endpoints of an edge and check that both reservations succeed in the commit component. Indeed this is how we implement Minimum Spanning Forest, after sorting the edges. However there is an optimization that can be made with spanning forests that involves only requiring one of the reservations to succeed. This increases the probability a commit will succeed and reduces the cost. This approach returns a different forest than the sequential version but is internally deterministic for a fixed schedule of prefix sizes.

The C++ code is given in Figure 4. For an iterate i corresponding to the edge $E[i]$ the reserve component does a `find` on each endpoint (as in the sequential algorithm) returning u and v (w.l.o.g., assume $u \leq v$). If $u = v$, the edge is within a component and can be dropped returning 0 (false)[1], otherwise the algorithm reserves v with the index i ($R[v].\mathtt{reserve}(i)$). The commit component for index i performs a $R[v].\mathtt{check}(i)$ to see if its reservation succeeded. If it has, it links v to u and otherwise the commit fails. At the end of the algorithm the edges $E[i]$ in the spanning tree can be identified as those where $R[i] \neq \bot$. The only difference from the sequential algorithm is that after determining that an edge goes between components instead of doing the union immediately it reserves one of the two sides. It later comes back to check that the reservation succeeded and if so does the union (link).

We note that in a round the reservation guarantees that only one edge (the highest priority) will link a vertex v to another vertex. This is the condition required in Section 3 for commutativity of `link`. Also because the `link` and `find` are in different phases they are never logically parallel, as required. Finally we note that because we link higher to lower vertex numbers the algorithm will never create a cycle. In this algorithm our code sets `psize`, the size of the prefix, to be $.02|E|$ and we have observed that on our test graphs less than 10% of the reservations fail.

[1] If false is returned by `reserve`() then the iterate is dropped without proceeding to the commit.

```
enum FlType {LIVE, IN, OUT};

struct MISStep {
  FlType flag; vertex *V;
  MISStep(char* _F, vertex* _V) : flag(_F), V(_V) {}

  bool reserve(int i) {
    int d = V[i].degree;
    flag = IN;
    for (int j = 0; j < d; j++) {
      int ngh = V[i].Neighbors[j];
      if (ngh < i) {
        if (Fl[ngh] == IN) { flag = OUT; return 1;}
        else if (Fl[ngh] == LIVE) flag = LIVE; } }
    return 1; }

  bool commit(int i) { return (Fl[i] = flag) != LIVE;}
};

void MIS(FlType* Fl, vertex* V, int n, int psize)
  speculative_for(MISStep(Fl, V), 0, n, psize);
}
```

Figure 5. C++ code for maximal independent set using deterministic reservations.

Minimum Spanning Forest: We use a parallel variant of Kruskal's algorithm. The idea of Kruskal's algorithm is to sort the edges and then add them one-by-one using disjoint sets as in the spanning forest code. We can therefore use deterministic reservations prioritized by the sorted order to insert the edges. Unlike the spanning forest described above, however, we need to reserve both endpoints of an edge to guarantee the edges are inserted in "sequential" order. However, during the commit component we only need that one of the two endpoint succeeds because to commute `link` only requires that one of the two arguments is unique. If v succeeds, for example, then we can use $\mathtt{link}(v, u)$. Note this is still internally deterministic because which endpoints succeed is deterministic. In our code we also make a further optimization: We sort only the smallest k edges ($k = \min(|E|, 4|V|/3)$ in our experiments) and run MSF on those, so that the remaining edges can be filtered out avoiding the need to sort them all. The sequential algorithm to which we compare our code does the same optimization.

Maximal Independent Set: Sequentially the maximal independent set can easily be calculated using the greedy method: loop over the vertices in an arbitrary order and for each vertex if no neighbors belong in the set add it to the set. There is a particularly simple way to implement this with deterministic reservations without even requiring an explicit `reserve`. The C++ code based on our interface is given in Figure 5 and an example of how the algorithm proceeds is shown in Figure 6. The `struct MISStep` defines the code for the reserve and commit components for each loop iteration. The array `V` stores for each of the `n` vertices its degree and a pointer to an array of neighbors. The array `Fl` keeps track of the status of each vertex—`LIVE` indicates it is still live, `IN` indicates it is done and in the set, and `OUT` indicates it is done and not in the set (a neighbor is in the set). The reserve phase for each iteration `i` loops over the neighbors of `V[i]` and sets a local variable `flag` as follows:

$$\mathtt{flag} = \begin{cases} \mathtt{OUT} & \text{any earlier neighbor is } \mathtt{IN} \\ \mathtt{LIVE} & \text{any earlier neighbor is } \mathtt{LIVE} \\ \mathtt{IN} & \text{otherwise} \end{cases}$$

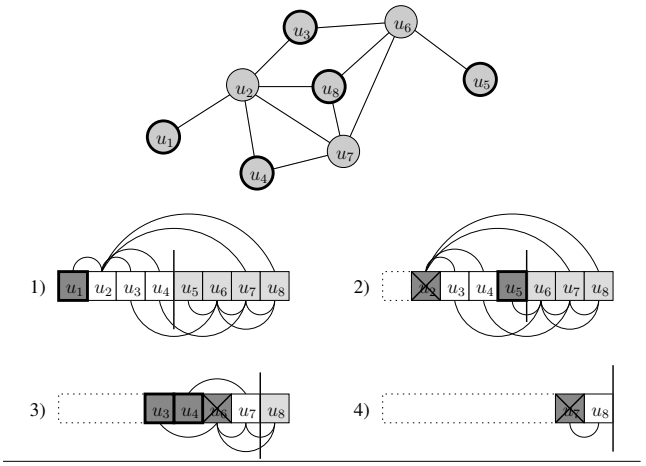


Figure 6. A sample graph and an execution of deterministic reservations for finding a maximal independent set. Here, the subscript of a node corresponds to its priority in the deterministic reservations. The prefix size is chosen to be 4. (1) shows the initial graph in priority order, and (2)-(4) show subsequent rounds of the algorithm. The vertical line indicates the end of the current prefix. Dark-gray nodes are those that become IN or OUT during that round: nodes with a thick border are IN and accepted into the MIS, and nodes with an "X" are OUT as they have a neighbor already in the MIS. For example, u_1 is the only node accepted into the MIS during the first round. Similarly, u_2 becomes OUT in the second round as it has a neighbor already in the MIS (namely, u_1). White nodes are those belonging to the current prefix that remain LIVE. For example, in the first round u_2, u_3, and u_4 all have a higher priority neighbor in the same prefix and remain live. Only nodes that survive the previous round (LIVE nodes) are displayed in the array and part of the current prefix, so u_5 is skipped in (3). Nodes in the MIS are also shown with thick border in the graph.

The second case corresponds to a conflict since for an earlier neighbor it is not yet known if it is IN or OUT. The commit phase for iteration `i` simply copies the local `flag` to `Fl[i]`. Since `Fl` is only read in the reserve phase and only written (to location `i`) in the commit phase, all operations commute.

Triangle Ray Intersect: We use a k-d tree with the surface area heuristic (SAH) [34] to store the triangles. Our algorithm is similar to the parallel algorithm discussed in [16] and makes use of divide-and-conquer and heavy use of `scan` and `filter`.

Suffix Array: We use a parallel variant of the algorithm of Karkkainen and Sanders [29]. It uses sorting and merging as subroutines, which involves nesting, but otherwise only makes use of `reduce`, `scan` and `filter`.

Delaunay Triangulation: We use a Boyer-Watson style incremental Delaunay algorithm with deterministic reservations. The points are used as the elements. To reduce contention, the prefix is always selected to be smaller than the current size of the mesh. The algorithm therefore starts out sequentially until enough points have been added. The reserve component of the code, for a point p, identifies all triangles that contain p in their circumcircle, often referred to as the hole for p. Adding p requires removing the hole and replacing it with other triangles. The reserve component therefore reserves all vertices around the exterior of the hole. The majority of the work required by a point p is in locating p in the mesh and then identifying the triangles in the hole. The commit component checks if all the reserved vertices of the mesh have succeeded, and if so, removes the hole and replaces it with triangles surrounding p and filling the hole. The reservations ensure that all modifications to the mesh commute since the triangles in the mesh only interact if they share a vertex. In fact, reserving the edges of the hole would be sufficient and reduce contention, but our mesh implementation has no data structures corresponding to edges on which to reserve. For efficiently locating a point p in the mesh we use the nearest neighbor structure described below.

Delaunay Refine: This algorithm uses the same routines for inserting points as the Delaunay triangulation. However, it does not need a point location structure but instead needs a structure to store the bad triangles. We use dynamic map for this purpose.

N-body: We use a parallel variant of the Callahan-Kosaraju algorithm [13]. This is a variant of Greengard and Rothkin's well-known FMM algorithm but allows more flexibility in the tree structure. The algorithm makes use of traditional nested parallelism with divide-and-conquer, as well as `reduce` and `scan`.

K-Nearest Neighbors: We use a quad- and oct-tree built over all input points for 2d and 3d inputs, respectively. As with the k-d tree used in triangle-ray intersection, the tree is built using only divide-and-conquer and nested parallelism. Once built, the tree is static and used only for queries of the points.

6. Experimental Results

We ran our experiments on a 32-core (with hyper-threading) Dell PowerEdge 910 with 4×2.26GHZ Intel 8-core X7560 Nehalem Processors, a 1066MHz bus, and 64GB of main memory. The parallel programs were compiled using the `cilk++` compiler (build 8503) with the `-O2` flag. The sequential programs were compiled using `g++` 4.4.1 with the `-O2` flag.

This section reports on the results for the benchmarks, as summarized in Table 2. We discuss six of the benchmarks in some detail, relating the performance to other published results. For each benchmark and given core count, the reported time for each input is the median time over three trials. We give only average timings over all inputs for the remaining benchmarks due to limited space.

For *Comparison Sort*, we used a variety of inputs all of length 10^7. This includes sequences of doubles in three distributions and two sequences of character strings. Both sequences of character strings are the same but in one the strings are allocated in order (*i.e.*, adjacent strings are likely to be on the same cache line) and in the other they are randomly permuted. We compare our internally deterministic sample sort to three other sorts: the standard template library (STL) sort, the parallel STL sort [41], and a simple divide-and-conquer quicksort that makes parallel recursive calls but partitions the keys sequentially. The results are summarized in Figure 7(a) and Table 3(a). Due to the cache-friendly nature of our algorithm, on average it is more efficient than any of the algorithms even on one core. However it is not quite as fast on the double-precision values since there the cache effects are less significant. As expected the quicksort with serial partitioning does not scale.

For *Remove Duplicates*, our inputs were all of length 10^7. We use both sequences of integers drawn from three distributions and sequences of integers corresponding to character strings. As shown in Figure 7(b) and Table 3(b), our parallel internally deterministic algorithm obtains good speedup (over 24x on 64 threads) and outperforms the serial version using 2 or more threads. On a single thread, it is only slightly slower than the serial version.

For *Breadth First Search (BFS)*, and all of the graph algorithms, we use three types of graphs: random graphs, grid graphs, and rMat graphs [14]. The rMat graphs have a power-law distribution of degrees. All edge counts are the number of undirected edges—we actually store twice as many since we store the edge in each direction. We compare our internally deterministic BFS to a serial version and a nondeterministic version (ndBFS). The results are summarized in Figure 7(c) and Table 3(c). Our nondeterministic

Application Algorithm	1 thread	64 threads (32h)	Speedup
Comparison Sort			
serialSort	3.581	–	–
*stlParallelSort	3.606	0.151	23.88
sampleSort	2.812	0.089	31.6
quickSort	3.043	0.68	4.475
Remove Duplicates			
serialHash	0.882	–	–
deterministicHash	1.034	0.042	24.62
Breadth First Search			
serialBFS	3.966	–	–
**ndBFS	5.4	0.28	19.29
deterministicBFS	7.136	0.314	22.73
**LS-PBFS†	4.357	0.332	13.12
Spanning Forest			
serialSF	2.653	–	–
deterministicSF	6.016	0.326	18.45
**Galois-ST$	12.39	1.136	10.91
Minimum Spanning Forest			
serialMSF	8.41	–	–
parallelKruskal	14.666	0.785	18.68
Maximal Independent Set			
serialMIS	0.501	–	–
**ndMIS	1.649	0.056	29.45
deterministicMIS	0.782	0.054	14.48
Triangle Ray Intersect			
kdTree	8.7	0.45	19.33
Suffix Array			
parallelKS	13.4	0.785	17.07
Delaunay Triangulation			
serialDelaunay	56.95	–	–
deterministicDelaunay	80.35	3.87	20.76
*Galois-Delaunay	114.116	39.36	2.9
Delaunay Refine			
deterministicRefine	103.5	6.314	16.39
**Galois-Refine‡	81.577	5.201	15.68
N-body			
parallelCK	122.733	5.633	21.79
K-Nearest Neighbors			
octTreeNeighbors	37.183	3.036	12.25

Table 2. Weighted average of running times (seconds) over various inputs on a 32-core machine with hyper-threading (32h). A "*" indicates an internally nondeterministic implementation and a "**" indicates an externally (and hence internally) nondeterministic implementation. All other implementations are internally deterministic. †LS-PDFS does not generate the BFS tree, while our programs do. $Galois-ST generates only a spanning tree, while our code generates the spanning forest. ‡Galois-Refine does not include the time for computing the triangle neighbors and initial bad triangles at the beginning while our code does (takes 10-15% of the overall time).

version is slightly faster than the deterministic version due to the fact that it avoids the second phase when processing each round. We have also compared times to published results. We ran the parallel breadth-first search algorithm from [32] on our graphs and our performance is very close to theirs (their algorithm is labeled LS-PBFS in our tables and figures). Our performance is 5 to 6 times faster than the times reported in [25] (both for 1 thread and 32 threads), but their code is written in Java instead of C++ and is on a Sun Niagara T2 processor which has a clock speed of 1.6Ghz instead of 2.26Ghz so it is hard to compare.

For *Minimum Spanning Forest (MSF)*, we compare our internally deterministic algorithm to an optimized version of Kruskal's serial algorithm (see Section 5). Our results are shown in Figure 7(d) and Table 3(d). Our code is about 1.7x slower on a single thread. We also compared our times to the parallel version of Boruvka's algorithm from the recent C++ release (2.1.0) of the Galois benchmark suite [38] (labeled as Galois-Boruvka in our table) on our inputs. Their code did not terminate in a reasonable amount of time on the random and rMat graphs; for the 2D-grid graph, our code is much faster and achieves much better speedup than their algorithm.

For *Maximum Independent Set (MIS)*, we compare our internally deterministic algorithm to the very simple and efficient serial algorithm and a nondeterministic version that uses locks (with compare-and-swap) before adding a vertex to the set. The results are summarized in Figure 7(e) and Table 3(e). As the experiments show, for this problem the deterministic algorithm is actually faster than the nondeterministic one. This is presumably because the deterministic version can avoid the reservation as discussed in Section 5 and therefore has little overhead compared to the serial algorithm. On one thread the nondeterministic algorithm is about a factor of 1.6x slower than the serial algorithm. We view this as quite good given the simplicity of the serial code. We note that MIS is about 5-10x faster than BFS on the same size graph.

For *Delaunay Triangulation*, we use two point distributions: points distributed at random and points distributed with the Kuzmin distribution. The latter has a very large scale difference between the largest and smallest resulting triangles. We compare our internally deterministic algorithm to a quite optimized serial version. Our results are shown in Figure 7(f) and Table 3(f). On one core it is a factor of about 1.4 slower, but it gets good speedup. We compared our code to the implementations in the Galois benchmark suite [38] (labeled as Galois-Delaunay and Galois-Refine in our tables and figures), and our triangulation code is faster and achieves better speedup on the same machine. We note, however, that on the Delaunay refinement problem we achieve almost the same run time as the Galois benchmarks (after subtracting the time for computing the initial processing of triangles from our times, which is about 10-15% of the overall time, since this is not part of the timing in the Galois code). Since the time for the refinement code is dominated by triangle insertion and the code for triangulation is dominated by point location, it would appear that the reason for our improved performance is due to our point location, and triangle insertion performs about equally well.

7. Conclusion

This paper has provided evidence that internally deterministic programs can remain efficient and indeed even rival the best nondeterministic approaches. In fact, in the case of MIS, using deterministic reservations revealed that some synchronization overheads could be removed, thereby improving performance.

Our approach uses nested parallelism with commuting and linearizable parallel operations. We have not addressed the issue of how to verify that operations commute or are linearizable, but the techniques we use are simple enough that it is quite easy to reason about the correctness. For example in deterministic reservations a user only needs to verify that the operations within the reserve component and separately within the commit component commute. It should also be feasible to adapt efficient techniques for runtime race detection [15] to check for parallel non-commuting operations.

It would also be interesting to conduct an empirical study supporting the programmability and debuggability claims for internal determinism. We have provided evidence that the programs in this paper have short code descriptions, but we have not studied how natural these programs are to develop in the first place.

(a)

Comparison Sort Algorithm	10^7 random		10^7 exponential		10^7 almost sorted		10^7 trigram		10^7 trigram (permuted)	
	(1)	(32h)	(1)	(32h)	(1)	(32h)	(1)	(32h)	(1)	(32h)
serialSort	1.42	–	1.1	–	0.283	–	4.31	–	5.5	–
*stlParallelSort	1.43	0.063	1.11	0.057	0.276	0.066	4.31	0.145	5.57	0.236
sampleSort	2.08	0.053	1.51	0.042	0.632	0.028	3.21	0.095	3.82	0.131
quickSort	1.58	0.187	1.06	0.172	0.357	0.066	3.35	0.527	4.78	1.31

(b)

Remove Duplicates Algorithm	10^7 random		10^7 random (values up to 10^5)		10^7 exponential		10^7 trigram		10^7 trigram (permuted)	
	(1)	(32h)	(1)	(32h)	(1)	(32h)	(1)	(32h)	(1)	(32h)
serialHash	0.654	–	0.311	–	0.504	–	0.849	–	1.31	–
deterministicHash	0.895	0.037	0.419	0.019	0.658	0.026	0.997	0.046	1.45	0.052

(c)

BFS Algorithm	random local graph $n = 10^7$ $m = 5 \times 10^7$		rMat graph $n = 2^{24}$ $m = 5 \times 10^7$		3d grid $n = 10^7$	
	(1)	(32h)	(1)	(32h)	(1)	(32h)
serialBFS	4.14	–	4.86	–	2.9	–
**ndBFS	6.07	0.226	6.78	0.294	3.35	0.322
deterministicBFS	7.13	0.255	9.25	0.345	5.03	0.343
**LS-PBFS	4.644	0.345	5.404	0.426	3.023	0.225

(d)

MSF Algorithm	random local graph $n = 10^7$ $m = 5 \times 10^7$		rMat graph $n = 2^{24}$ $m = 5 \times 10^7$		2d grid $n = 10^7$	
	(1)	(32h)	(1)	(32h)	(1)	(32h)
serialMSF	8.47	–	11.2	–	5.56	–
parallelKruskal	14.3	0.78	19.7	1.08	10.0	0.49
*Galois-Boruvka†	–	–	–	–	35.128	7.159

(e)

MIS Algorithm	random local graph $n = 10^7$ $m = 5 \times 10^7$		rMat graph $n = 2^{24}$ $m = 5 \times 10^7$		2d grid $n = 10^7$	
	(1)	(32h)	(1)	(32h)	(1)	(32h)
serialMIS	0.447	–	0.669	–	0.388	–
**ndMIS	1.49	0.051	2.11	0.068	1.35	0.042
deterministicMIS	0.665	0.047	1.09	0.07	0.593	0.041

(f)

Delaunay Triangulation Algorithm	2d in cube $n = 10^7$		2d kuzmin $n = 10^7$	
	(1)	(32h)	(1)	(32h)
serialDelaunay	55.1	–	58.8	–
deterministicDelaunay	76.7	3.5	84.0	4.24
*Galois-Delaunay	110.705	39.333	117.527	36.302

Table 3. Running times (seconds) of algorithms over various inputs on a 32-core machine (with hyper-threading). A "*" indicates an internally nondeterministic implementation and a "**" indicates an externally (and hence internally) nondeterministic implementation. †Galois-Boruvka did not terminate in a reasonable amount of time for the first two inputs.

Acknowledgments. This work is partially supported by the National Science Foundation under grant number CCF-1018188, and by Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program.

References

[1] U. Acar, G. E. Blelloch, and R. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3), 2002. Springer.

[2] S. V. Adve and M. D. Hill. Weak ordering–a new definition. In *ACM ISCA*, 1990.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ACM ASPLOS*, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Usenix OSDI*, 2010.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *ACM OOPSLA*, 2009.

[6] G. E. Blelloch. Programming parallel algorithms. *CACM*, 39(3), 1996.

[7] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *IEEE FOCS*, 2007.

[8] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ACM ICFP*, 1996.

[9] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. In *ACM SPAA*, 2010.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel and Distributed Computing*, 37(1), 1996. Elsevier.

[11] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Usenix HotPar*, 2009.

[12] R. L. Bocchino, 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 *ACM POPL*, 2011.

[13] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1), 1995.

[14] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM SDM*, 2004.

[15] 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 *ACM SPAA*, 1998.

[16] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel SAH k-D tree construction. In *ACM High Performance Graphics*, 2010.

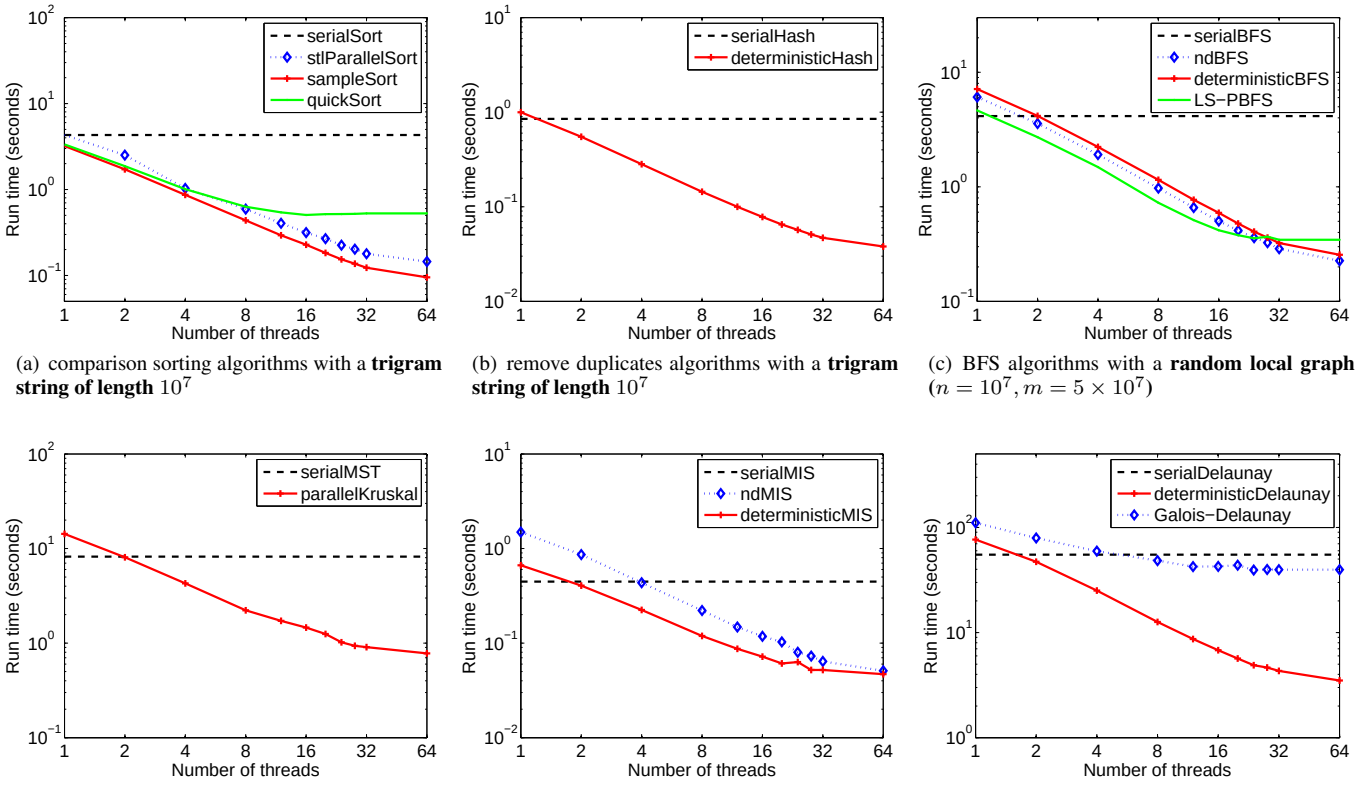
[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

[18] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[19] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ACM ASPLOS*, 2009.

[20] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A relaxed consistency deterministic computer. In *ACM ASPLOS*, 2011.

[21] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD 123, Dept. of Mathematics, Technological U., Eindhoven, 1965.



(a) comparison sorting algorithms with a **trigram string of length** 10^7

(b) remove duplicates algorithms with a **trigram string of length** 10^7

(c) BFS algorithms with a **random local graph** $(n = 10^7, m = 5 \times 10^7)$

(d) MST algorithms with a **weighted random local graph** $(n = 10^7, m = 5 \times 10^7)$

(e) MIS algorithms with a **random local graph** $(n = 10^7, m = 5 \times 10^7)$

(f) Delaunay Triangulation algorithms with a **2d in cube graph** $(n = 10^7)$

Figure 7. Log-log plots of running times on a 32-core machine (with hyper-threading). Our deterministic algorithms are shown in red.

[22] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ACM ISCA*, 1990.

[23] P. B. Gibbons. A more practical PRAM model. In *ACM SPAA*, 1989.

[24] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4), 1985.

[25] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *ACM PPoPP*, 2011.

[26] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *ACM PPoPP*, 2008.

[27] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[28] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? Free will to choose. In *IEEE HPCA*, 2011.

[29] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *EATCS ICALP*, 2003.

[30] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *ACM PLDI*, 2011.

[31] C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3), 2010. Springer.

[32] 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 *ACM SPAA*, 2010.

[33] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *ACM PPoPP*, 2012.

[34] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3), 1990. Springer.

[35] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM LOPLAS*, 1(1), 1992.

[36] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ACM ASPLOS*, 2009.

[37] S. S. Patil. Closure properties of interconnections of determinate systems. In J. B. Dennis, editor, *Record of the Project MAC conference on concurrent systems and parallel computation*. ACM, 1970.

[38] 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 *ACM PLDI*, 2011.

[39] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *ACM PLDI*, 2011.

[40] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM TOPLAS*, 19(6), 1997.

[41] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *Euro-Par*, 2007.

[42] G. L. Steele Jr. Making asynchronous parallelism safe for the world. In *ACM POPL*, 1990.

[43] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ACM ISCA*, 2000.

[44] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Computers*, 37(12), 1988.

[45] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ACM ISCA*, 2009.