Internally Deterministic Parallel Algorithms

Guy Blelloch

Carnegie Mellon University

Also: Jeremy Fineman, Phil Gibbons (Intel), Julian Shun, Harsha Vardham Simhadri, …

Partial Motivation

WoDet 2010: Seattle

Debate: can deterministic algorithms be as fast as nondeterministic ones?

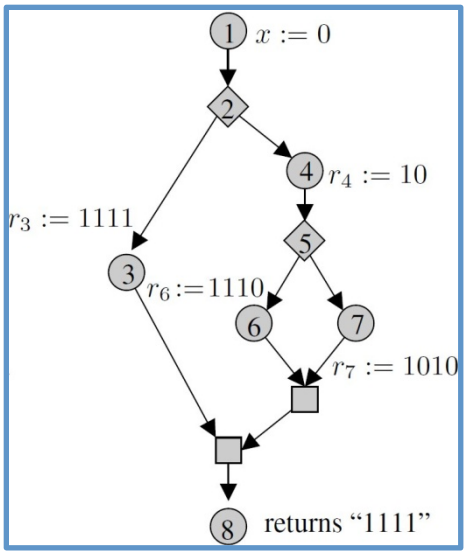
External vs. Internal Determinism [Emrath+Padua '88, Netzer+Miller '90]

- External: same input → same result
- Internal: same input → same "intermediate states" and same result

Internal Determinism

- Trace: a computation's final state, intermediate states, along with its control-flow DAG
- Internally deterministic: for any fixed input, all possible executions result in equivalent traces (with respect to some level of abstraction)
  - External determinism
  - Sequential semantics

Trace



Internally deterministic?

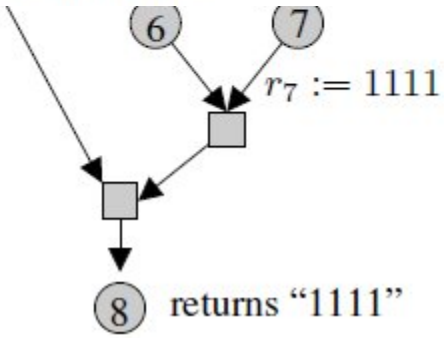
```
1.  x := 0
2.  in parallel do
3.      {   r3 := AtomicAdd(x, 1)     }
4.      {   r4 := AtomicAdd(x, 10)
5.          in parallel do
6.              {   r6 := AtomicAdd(x, 100)    }
7.              {   r7 := AtomicAdd(x, 1000)    }
        }
8.  return x
```

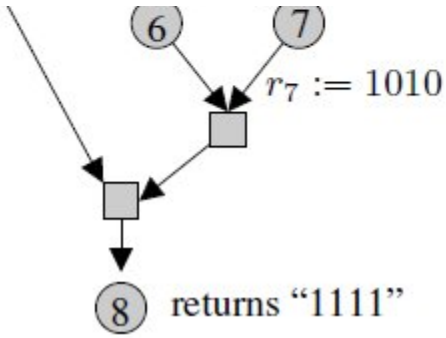

1. $x := 0$
2. **in parallel do**
3. $\{ \quad r_3 := \texttt{AtomicAdd}(x, 1) \quad \}$
4. $\{ \quad r_4 := \texttt{AtomicAdd}(x, 10)$
5. **in parallel do**
6. $\{ \quad r_6 := \texttt{AtomicAdd}(x, 100) \quad \}$
7. $\{ \quad r_7 := \texttt{AtomicAdd}(x, 1000) \quad \}$

$r_3 :=$

$\}$

8. `return` x





But!!

What does it mean for traces to be equivalent?

Emrath+Padua and Netzer+Miller: equal bit representation. Nice and simple.

But very restrictive: what about pointers from memory allocation?

Instead: abstract operations on data structures

- e.g. a dictionary
- equality is subtle

Encapsulation

Using abstract operations gives a technique to "encapsulate" non-determinism while still being internally deterministic.

- non-deterministic subroutines
- non-deterministic internal representations of data structures
- **more subtle**: non-deterministic implementations of linearizable but commutative data structures

Pros and Cons of Internal Determinism

?

- Debugging
- Composability
- Verification
- Performance analysis
- Sequential semantics

- Complicated code
- Performance penalty

Can Internally Deterministic Parallel Algorithms be fast and simple to code?

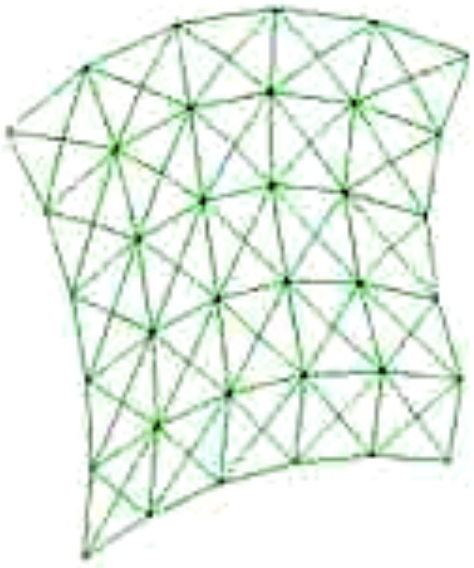
Internal Determinism in debugging

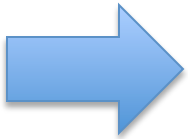
Need to be careful that measurement does not "perturb" the system.

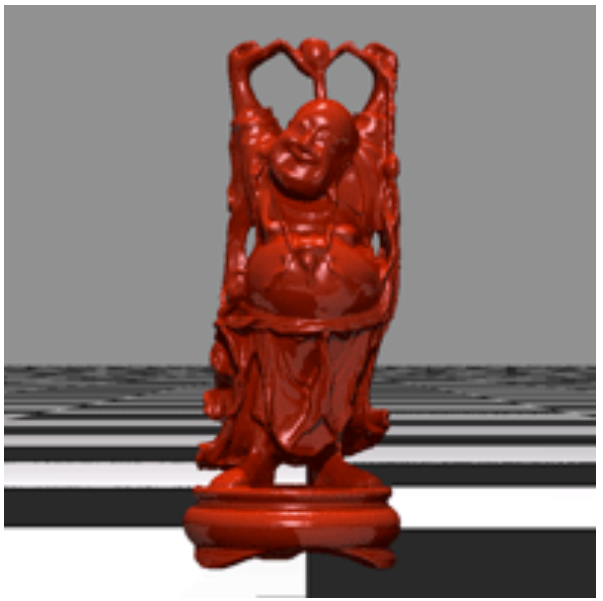
- Break points/Single stepping – can only make queries that are deterministic. Based on a particular sequential order.
- Adding observations ("print" statements) - need to collect them deterministically.

Problem Based Benchmarks

- Define a set of benchmarks in terms of **Input/ Output** behavior on specific inputs, and use them to compare solutions.







Input

Output

Many Existing Benchmarks

But none we know of match the spec

- **Code Based** : SPEC, Da Capo, PassMark, Splash-2, PARSEC, fluidMark
- **Application Specific**: Linpack, BioBench, BioParallel, MediaBench, SATLIB, CineBench, MineBench, TCP, ALPBench, Graph 500, DIMACS challenges
- **Method Based**: Lonestar
- **Machine analysis**: HPC challenge, Java Grande, NAS, Green 500, Graph 500, P-Ray, fluidMark

Preliminary Benchmarks I

Sequences	* Comparison Sorting
	* Removing Duplicates
	* Dictionary
Graphs	* Breadth First Search
	* Graph Separators
	* Minimum Spanning Tree
	* Maximal Independent Set
Geometry/ Graphics	* Delaunay Triangulation and Refinement
	* Convex Hulls
	* Ray Triangle Intersection (Ray Casting)
	Micropolygon Rendering

* finished

Preliminary Benchmarks II

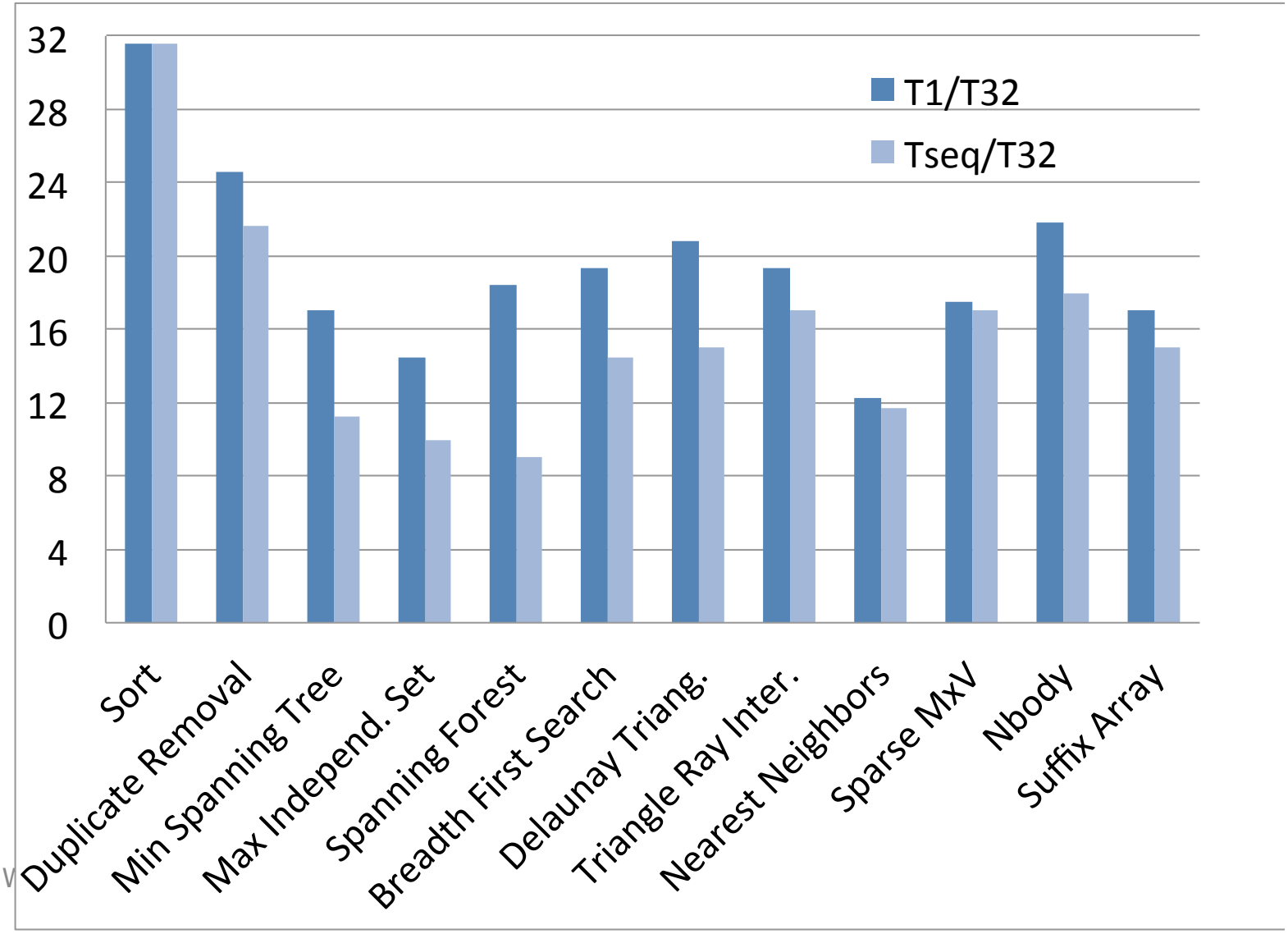
Machine Learning	* All Nearest Neighbors
	Support Vector Machines
	* K-Means
Text Processing	* Suffix Arrays
	Edit Distance
	String Search
Science	* Nbody force calculations
	Phylogenetic tree
Numerical	* Sparse Matrix Vector Multiply
	Sparse Linear Solve

* finished

Each Benchmark Consists of:

- A precise specification of the problem
- Specification of Input/Output file formats
- A set of input generators.
- Code for testing the results
- **Baseline sequential code**

How do the problems do on a modern multicore



Our Experiments

We coded up the 16 benchmarks from the PBBS with strict internal determinism trying various approaches.

Compared them with sequential code and non-deterministic code both in terms of runtime and code complexity.

Our Approach

Nested Parallelism +
Commutative/Linearizable operations (Steele '90)

Specific approaches:

- Functional programming
- History independent data structures
- Deterministic reservations

Written in g++ with just CAS, cilk_for, spawn, sync

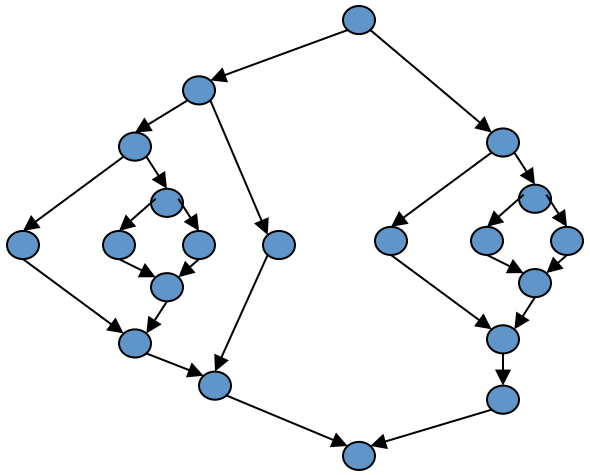
Nested Parallelism

We consider **nested parallel computations**

arbitrary nesting of fork-join and parallel loops

Has some important advantages:

- Good for caching
- Reduces scheduling overhead
- Supported by many languages
- Easy to analyze costs
- Makes it easier to verify code??



Commutativity

$f(S) \rightarrow S' \Rightarrow v$ *f converts from state S to S′ returning v*

f and g **commute** if for all S

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

$$g(S) \rightarrow S_g \Rightarrow v'_g \quad f(S_g) \rightarrow S_{gf} \Rightarrow v'_f$$

Implies: $v_f = v'_f$ and $v_g = v'_g$ and $S_{fg} = S_{gf}$

Linearizable

Concurrent operations appear to happen atomically.

Theorem: for a nested parallel computation if all parallel operations commute and are linearizable then the computation is internally deterministic.

Commutative Operations: Examples

- write-with-min, write-with-add
- dictionary
  - inserts commute
  - deletes commute
  - searches commute
- union-find structure
  - find(x) : commute
  - link(r_1, x_1) and link(r_2, x_2) commute if $r_1 \neq r_2$

Specific Approaches

- Functional programming
- History independent data structures
- Deterministic reservations

Functional programming
Suffix array
Comparison sort
N-body
K-nearest neighbors
Triangle ray intersect

History-independent data structures
Remove duplicates
Delaunay refinement

Deterministic reservations
Spanning forest
Minimum spanning forest
Maximal independent set
Breadth first search
Delaunay triangulation
Delaunay refinement

Internally Deterministic Problems

Functional programming
Suffix array
Comparison sort
N-body
K-nearest neighbors
Triangle ray intersect

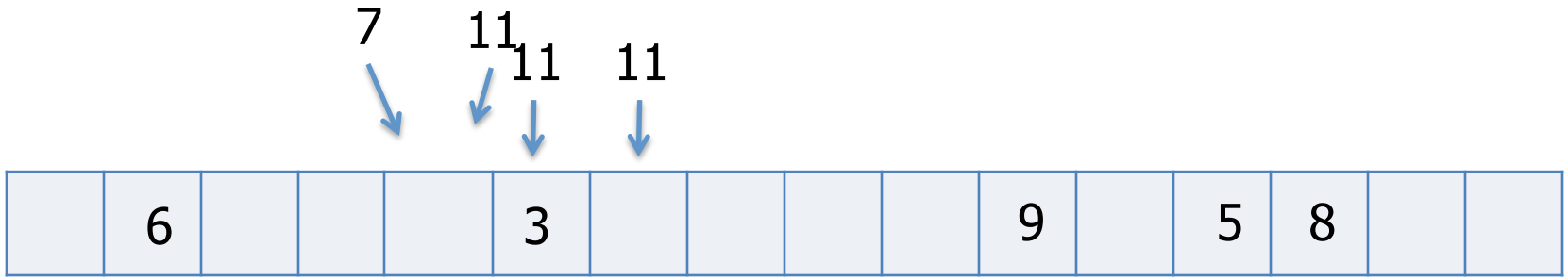
History-independent data structures
Remove duplicates
Delaunay refinement

Deterministic reservations
Spanning forest
Minimum spanning forest
Maximal independent set
Breadth first search
Delaunay triangulation
Delaunay refinement

Removing Duplicates

Using hashing:

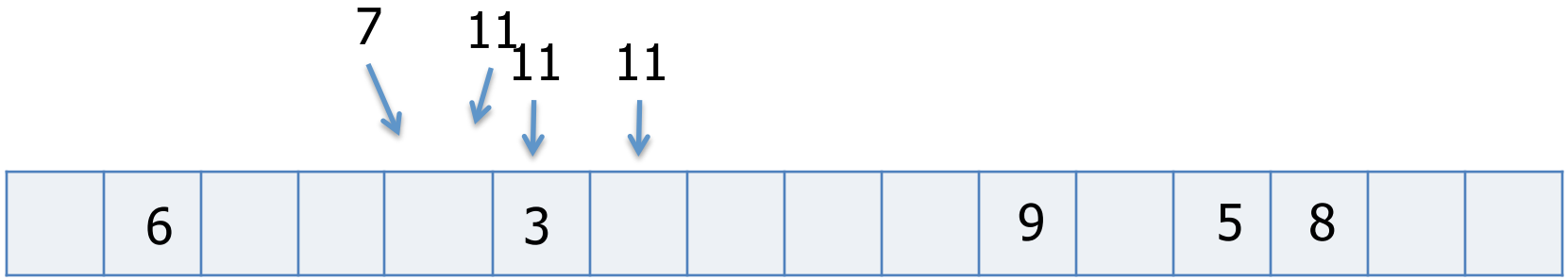
- Based on generic hash and comparison
- Problem: representation can depend on ordering. Also on which redundant element is kept.
- Solution: Use **history independent** hash table based on linear probing…once done inserting, representation is independent of order of insertion



Removing Duplicates

Using hashing:

- Based on generic hash and comparison
- Problem: representation can depend on ordering. Also on which redundant element is kept.
- Solution: Use **history independent** hash table based on linear probing…once done inserting, representation is independent of order of insertion



Internally Deterministic Problems

Functional programming
Suffix array
Comparison sort
N-body
K-nearest neighbors
Triangle ray intersect

History-independent data structures
Remove duplicates
Delaunay refinement

Deterministic Reservations
Spanning forest
Minimum spanning forest
Maximal independent set
Breadth first search
Delaunay triangulation
Delaunay refinement

Deterministic Reservations

Priority reserve all state that need to touch based on unique identifier. Only proceed if "won" on all reservations. Use write-with-min for reservation.

Example: speculative for

```
for (i=0; i<n; i++)
  f(i);
```

Want to simulate sequential order even with loop carried dependences which **don't commute.**

Speculative For

```
for (i=0; i<n; i++) f(i);
```

Converts to:

```
parallel_for (i=0; i<n; i++)
  reserve any shared vars with i
parallel_for (i=0; i<n; i++)
  if won all reservations, f(x)
  else mark for retry
```

Speculative For

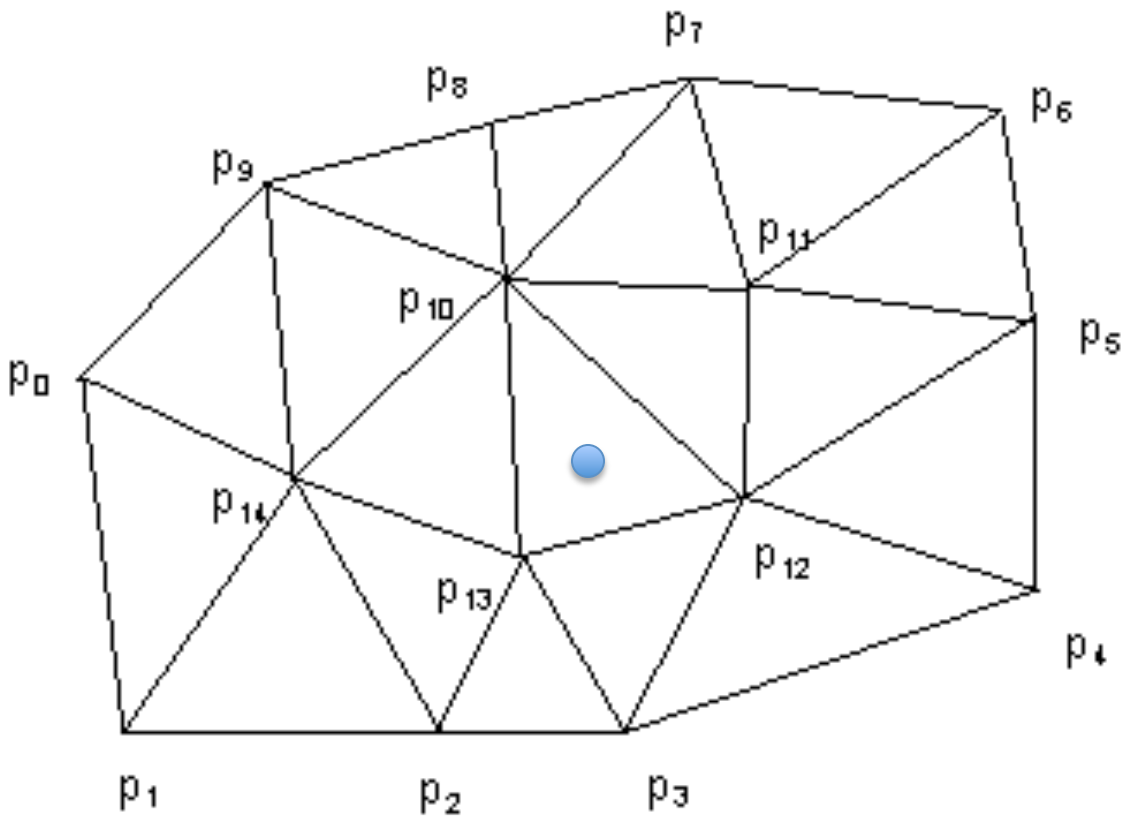
```
for (i=0; i<n; i++) f(i);
```

Or, more efficiently:

```
while (done < n)
  parallel_for (i=0; i<k; i++)
     reserve(I(i+done))
  parallel_for (i=0; i<n; i++)
     if succeed f(I(i+done))
  pack remaining indices into I
```

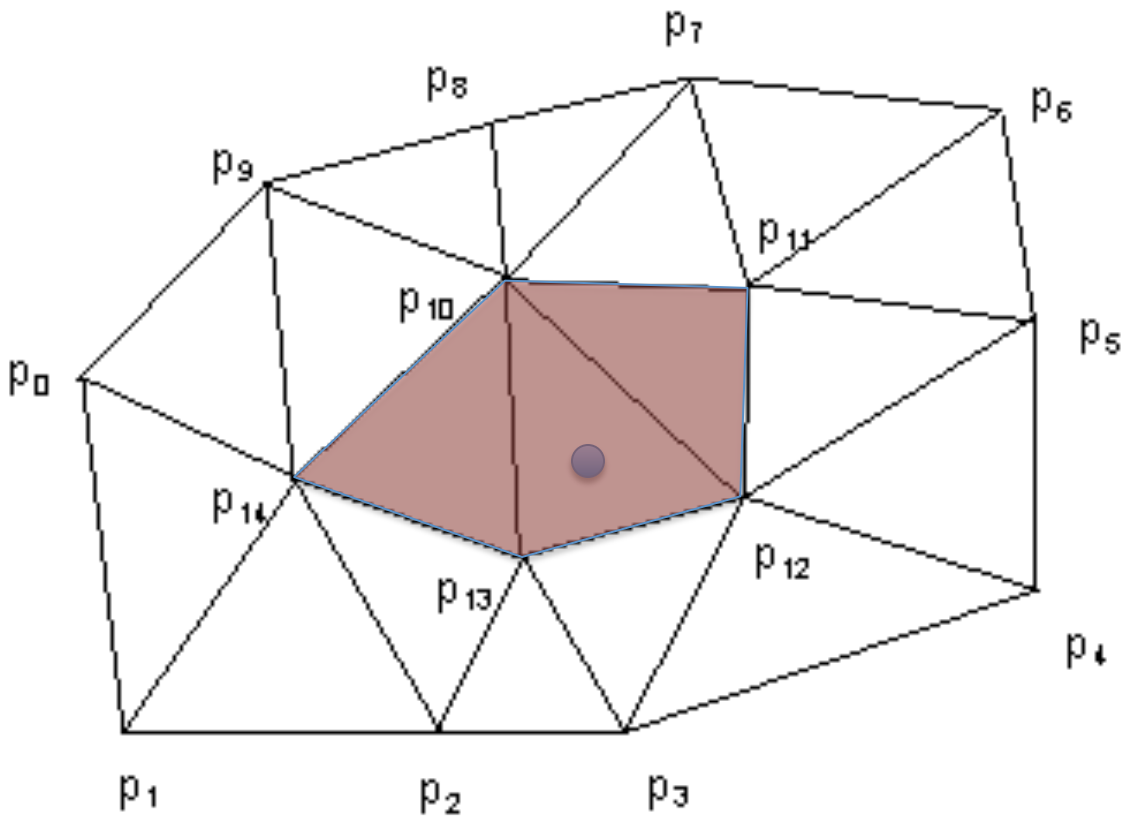
Delaunay Triangulation

- Adding points deterministically



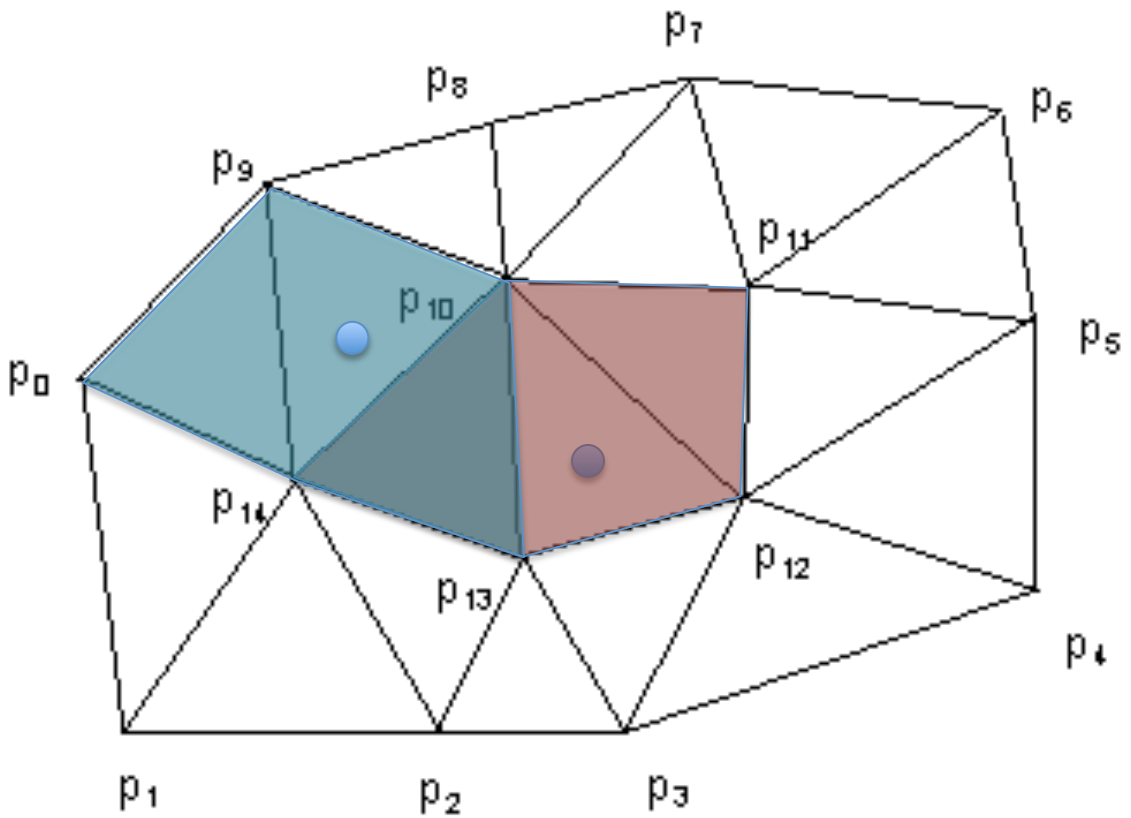
Delaunay Triangulation

- Adding points deterministically



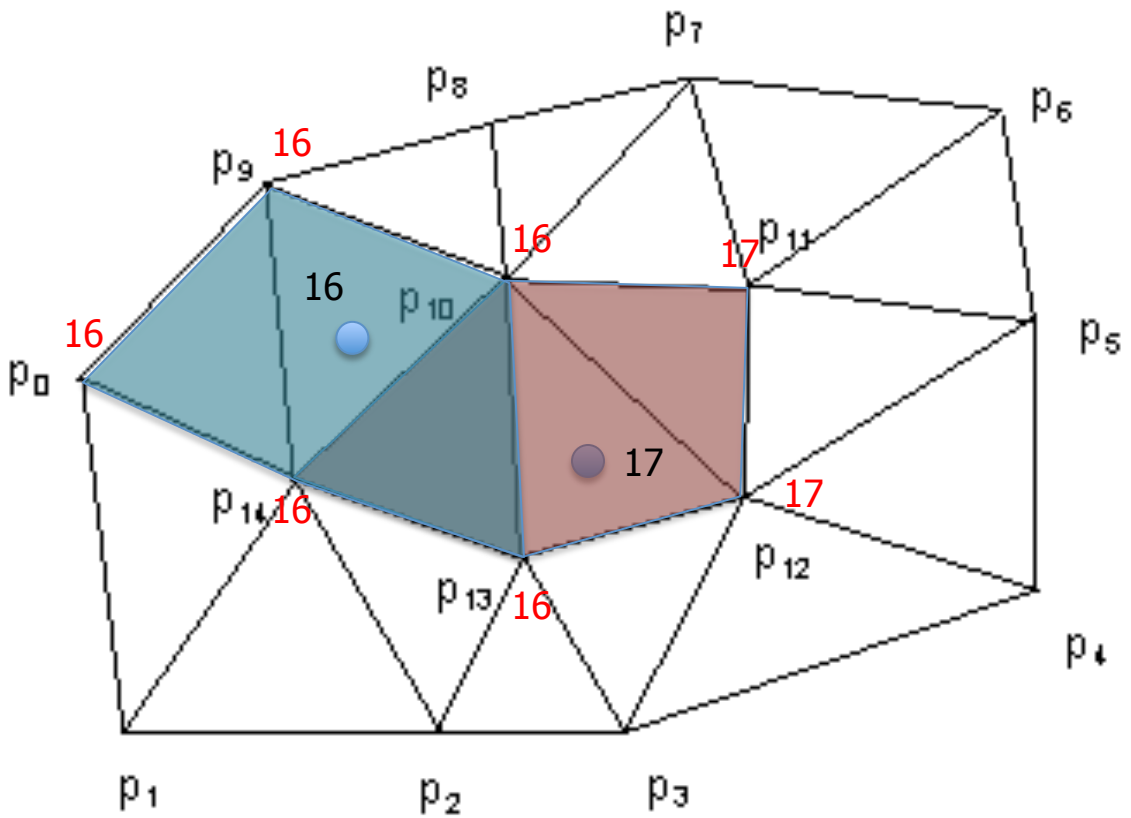
Delaunay Triangulation

- Adding points deterministically



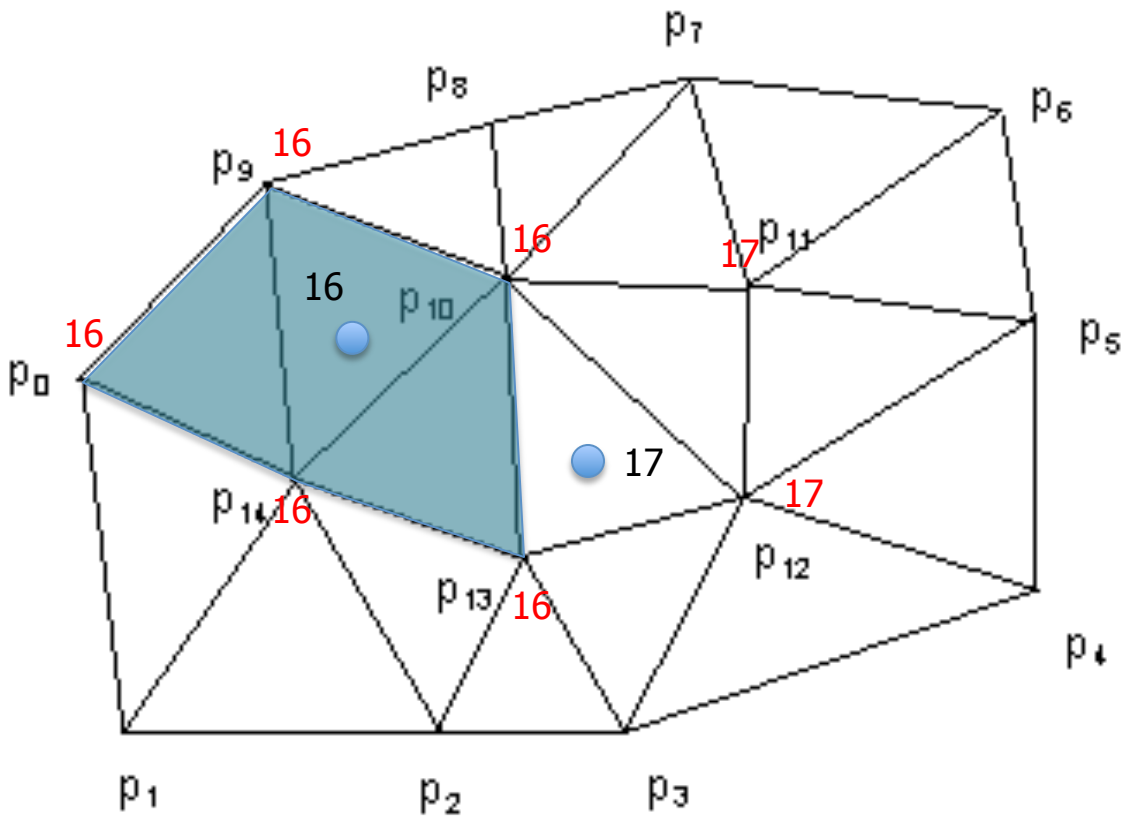
Delaunay Triangulation

- Adding points deterministically



Delaunay Triangulation

- Adding points deterministically



Deterministic Reservations

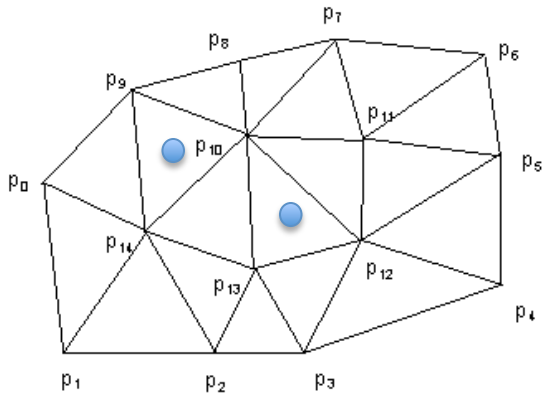
Generic framework

```
iterates  = [1,…,n];
while(iterates remain){

        Phase 1: in parallel, all i in first P iterates
                        call reserve(i);

        Phase 2: in parallel, all i in first P iterates
                        call commit(i);

        Remove successfully committed i's from
            iterates;
}
```

- Which iterates successfully commit is deterministic.

Delaunay triangulation

```
iterates:  points to be added

reserve(i){
      find cavity;
      reserve points in cavity;
}

commit(i){
      check reservations;
      if(all reservations successful){
            add point and triangulate;
      }
}
```



Internally Deterministic Problems

Functional programming
Suffix array
Comparison sort
N-body
K-nearest neighbors
Triangle ray intersect

History-independent data structures
Remove duplicates
Delaunay refinement

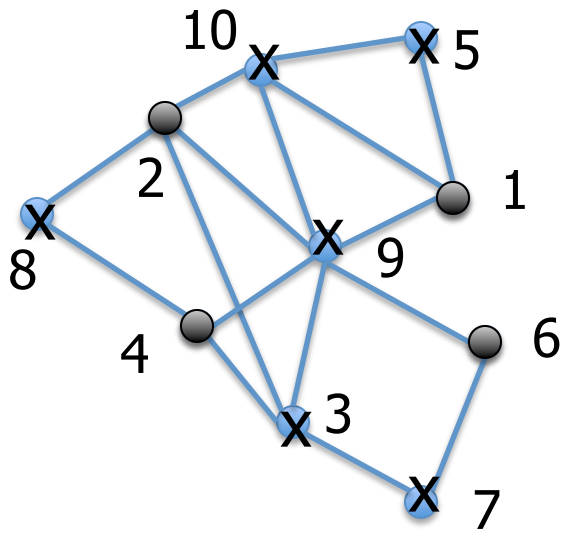
Deterministic reservations
Spanning forest
Minimum spanning forest
Maximal independent set
Breadth first search
Delaunay triangulation
Delaunay refinement

Maximal Independent Set

Important substep in Graph Coloring

Sequential algorithm:

```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```



Maximal Independent Set

Sequential algorithm:

```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```

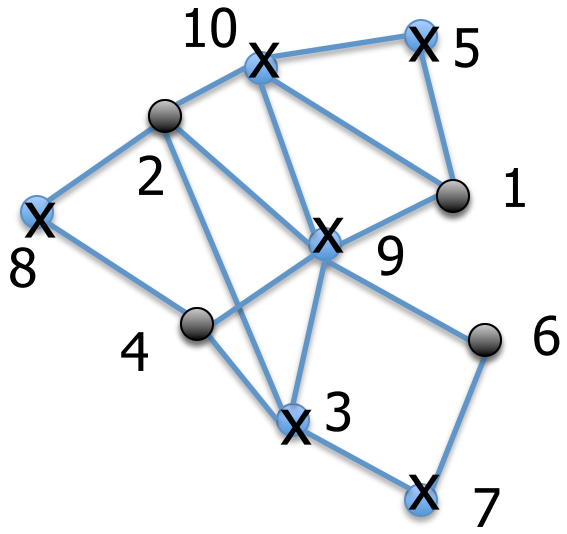
Very efficient: most edges not even visited, simple loops

About 7x faster than sorting m edges

Maximal Independent Set

Same algorithm: with parallel speculation

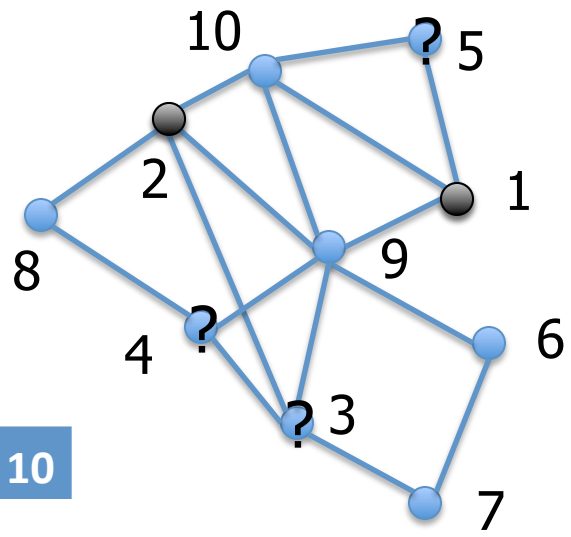
```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```



Maximal Independent Set

same algorithm: with speculation on prefix

```
for each u in V : S[u] = Remain
for each u in V
    if for all v in N(u), v < u, S[v] = Out
    then S[u] = In
    else S[u] = Out
```



1	2	3	4	5	6	7	8	9	10

MIS Parallel Code

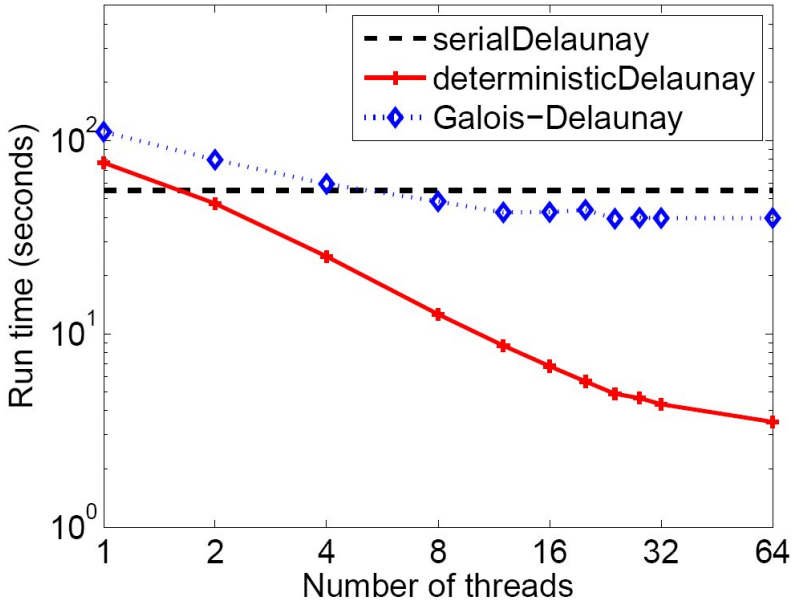
```
struct MISStep {
  bool reserve(int i) {
    int d = V[i].degree;
    flag = IN;
    for (int j = 0; j < d; j++) {
      int ngh = V[i].Neighbors[j];
      if (ngh < i) {
        if (Fl[ngh] == IN) { flag = OUT; return 1;}
        else if (Fl[ngh] == LIVE) flag = LIVE; } }
    return 1; }

  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);}
```

Experimental Results

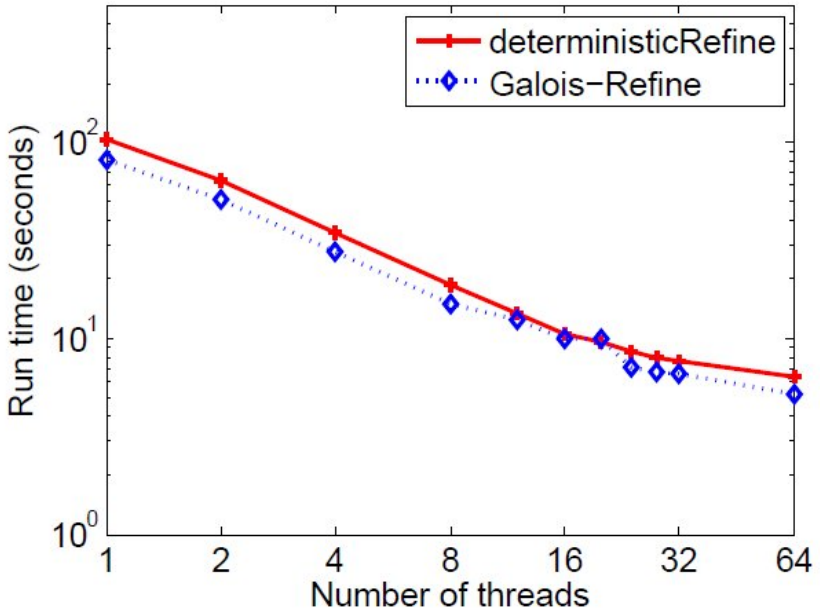
- We ran experiments comparing our deterministic implementations with serial and nondeterministic ones
- We used a 32-core (with hyper-threading) machine with 4 Intel X7560 Nehalem processors
- Used inputs drawn from a variety of distributions
- All codes are between 20 and 500 lines

Experimental Results

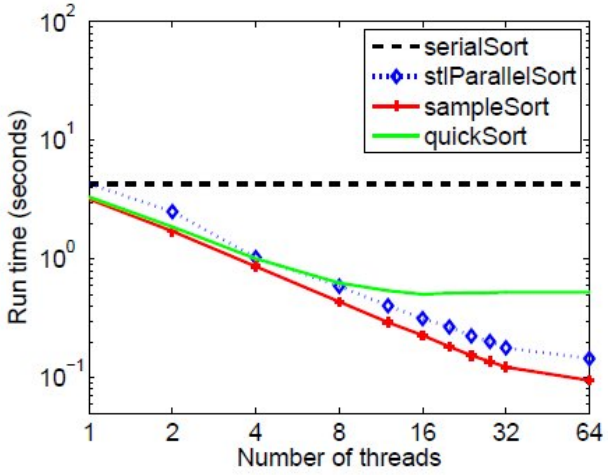
Delaunay Triangulation



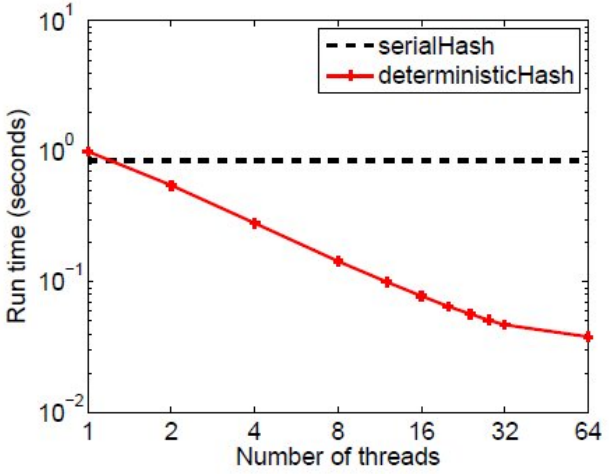
Delaunay Refinement



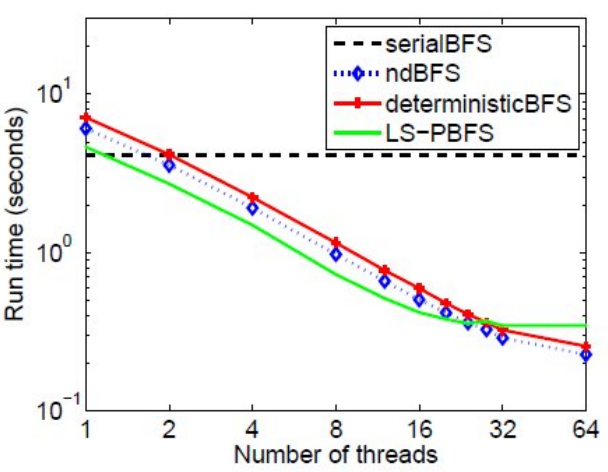
Experimental Results



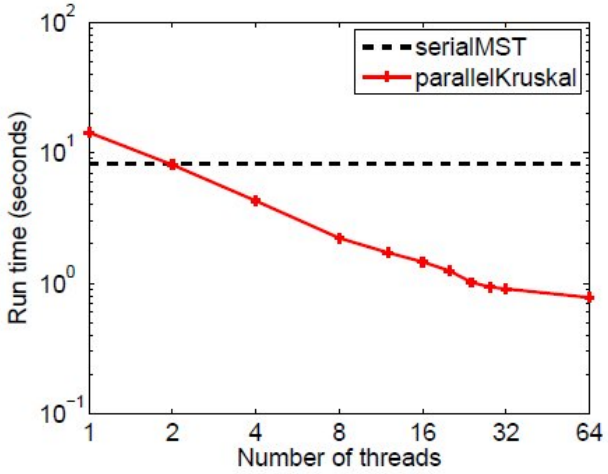
(a) comparison sorting algorithms with a **trigram string of length** 10^7



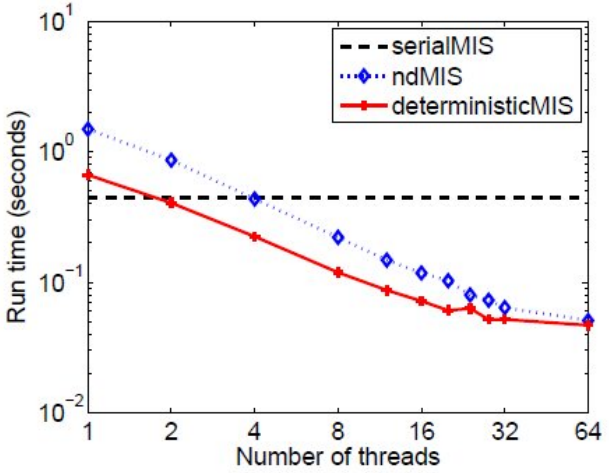
(b) remove duplicates algorithms with a **trigram string of length** 10^7



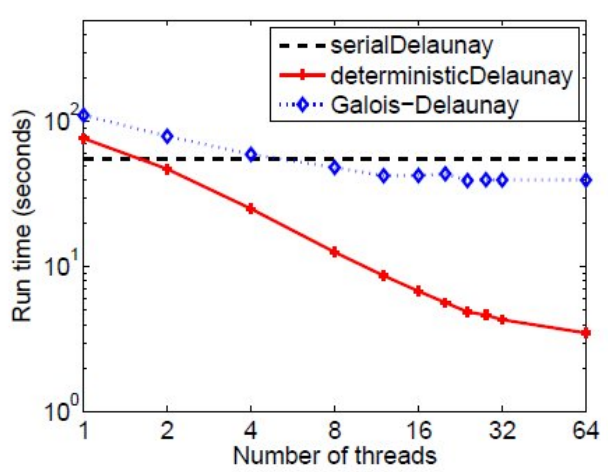
(c) BFS algorithms with a **random local graph** $(n = 10^7, m = 5 \times 10^7)$



(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** $(n = 10^7)$

Experimental Results

- Up to 31.6x speedup on 32 processors
- Competitive with nondeterministic solutions
- On a single processor, our deterministic implementations are only 1–2.5x slower than their sequential counterparts

Conclusions

- Internal Determinism can be fast
- Some simple techniques can help
  - History independence
  - Deterministic reservations
  - Functional programming

Open questions:

- What types of problems truly benefit from a nondeterministic solution?
- Safe commutativity?

`http://www.cs.cmu.edu/~pbbs`