

Multi-core Computing

Lecture 3

MADALGO Summer School 2012
Algorithms for Modern Parallel and Distributed Models

Phillip B. Gibbons
Intel Labs Pittsburgh

August 22, 2012

Multi-core Computing Lectures:

Progress-to-date on Key Open Questions

- **How to formally model multi-core hierarchies?**
- **What is the Algorithm Designer's model?**
- **What runtime task scheduler should be used?**
- **What are the new algorithmic techniques?**
- **How do the algorithms perform in practice?**

Lecture 1 & 2 Summary

- **Multi-cores: today, future trends, challenges**
- **Computations & Schedulers**
- **Cache miss analysis on 2-level parallel hierarchy**
- **Low-depth, cache-oblivious parallel algorithms**

- **Modeling the Multicore Hierarchy**
- **Algorithm Designer's model exposing Hierarchy**
- **Quest for a Simplified Hierarchy Abstraction**
- **Algorithm Designer's model abstracting Hierarchy**
- **Space-Bounded Schedulers**

Lecture 3 Outline

- **Cilk++**
- **Internally-Deterministic Algorithms**
- **Priority-write Primitive**
- **Work Stealing Beyond Nested Parallelism**
- **Other Extensions**
 - False Sharing
 - Work Stealing under Multiprogramming
- **Emerging Memory Technologies**

Multicore Programming using Cilk++

- Cilk extends the C language with just a *handful* of keywords
- Every Cilk program has a *serial semantics*
- Not only is Cilk fast, it provides *performance guarantees* based on performance abstractions
- Cilk is *processor-oblivious*
- Cilk's *provably good* runtime system automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling

Intel® Cilk™ Plus

Cilk++ Example: Fibonacci

```
int fib (int n) {  
  if (n<2) return (n);  
  else {  
    int x,y;  
    x = fib(n-1);  
    y = fib(n-2);  
    return (x+y);  
  }  
}
```

C elision

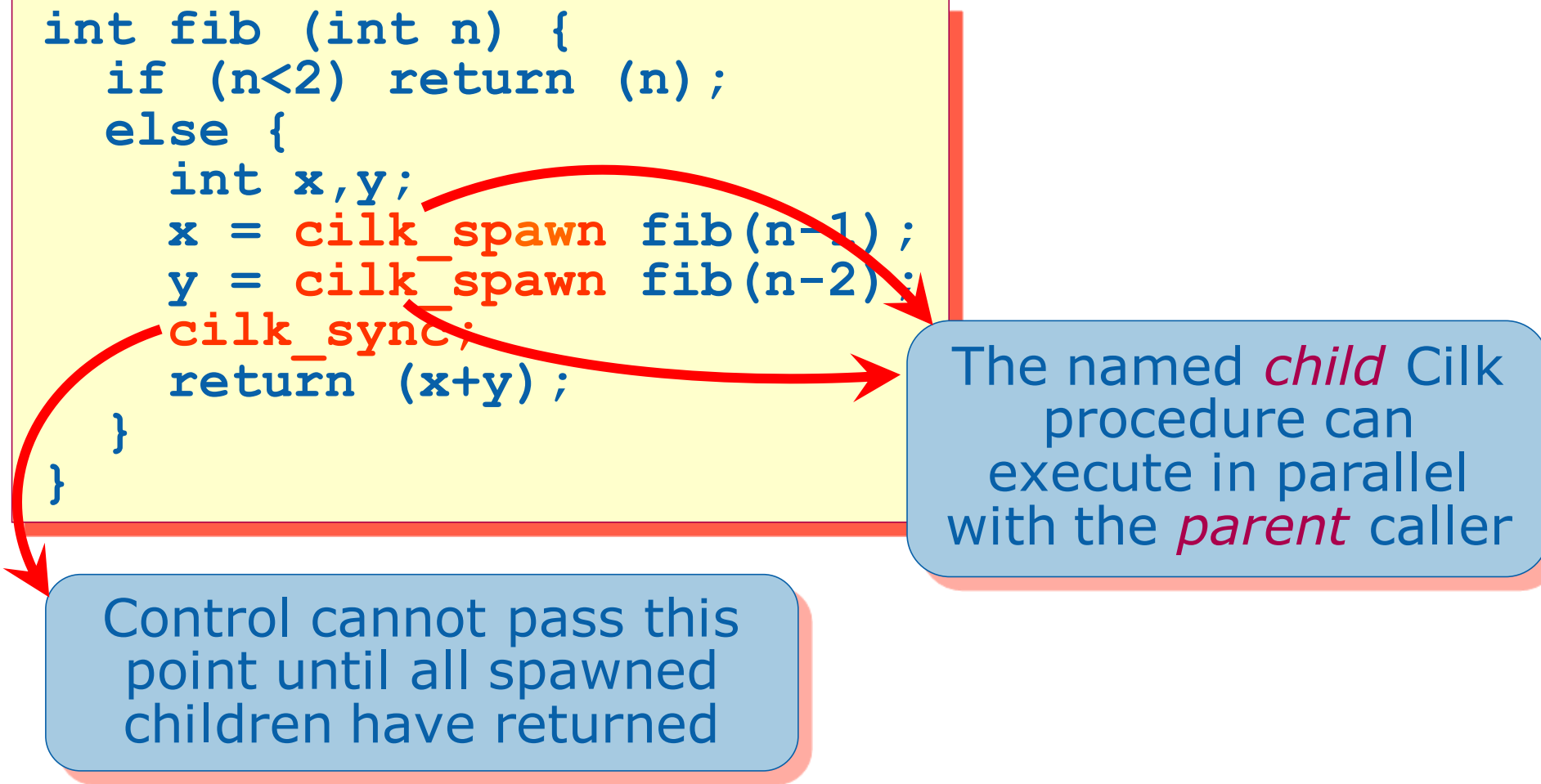
Cilk code

```
int fib (int n) {  
  if (n<2) return (n);  
  else {  
    int x,y;  
    x = cilk_spawn fib(n-1);  
    y = cilk_spawn fib(n-2);  
    cilk_sync;  
    return (x+y);  
  }  
}
```

Cilk is a *faithful* extension of C. A Cilk program's *serial elision* is always a legal implementation of Cilk semantics. Cilk provides *no* new data types.

Basic Cilk++ Keywords

```
int fib (int n) {  
    if (n<2) return (n);  
    else {  
        int x,y;  
        x = cilk_spawn fib(n-1);  
        y = cilk_spawn fib(n-2);  
        cilk_sync;  
        return (x+y);  
    }  
}
```



The named *child* Cilk procedure can execute in parallel with the *parent* caller

Control cannot pass this point until all spawned children have returned

Useful macro: `cilk_for`
for recursive spawning of parallel loop iterates

Nondeterminism in Cilk

- ▶ **Cilk** encapsulates the nondeterminism of scheduling, allowing average programmers to write deterministic parallel codes using only **3 keywords** to indicate logical parallelism
- ▶ The **Cilkscreen** race detector offers **provable guarantees of determinism** by certifying the absence of determinacy races
- ▶ **Cilk's reducer hyperobjects** encapsulate the nondeterminism of updates to nonlocal variables, yielding deterministic behavior for parallel updates
 - ▶ **See next slide**

Summing Numbers in an Array using `sum_reducer` [Frigo et al. '09]

```
int compute(const X& v);
int cilk_main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...
    sum_reducer<int> result(0);
    cilk_for (std::size_t i = 0; i < n; ++i)
        result += compute(myArray[i]);

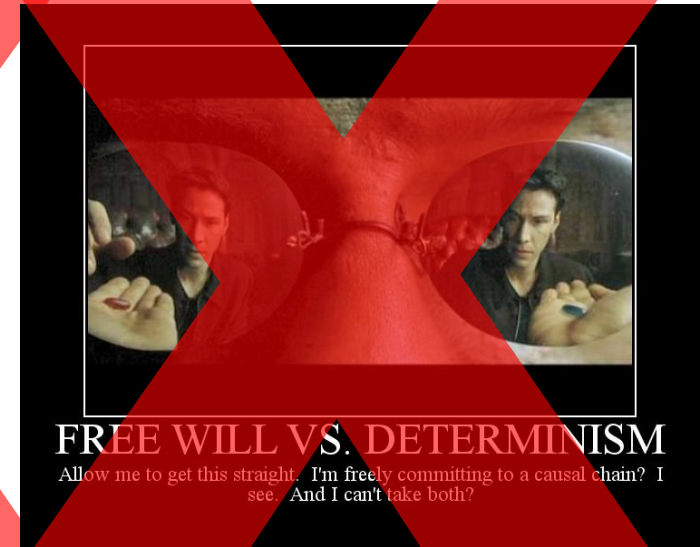
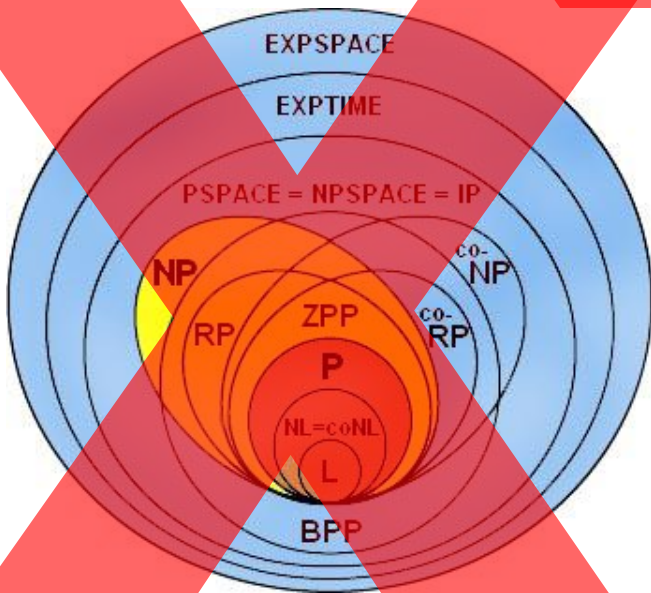
    std::cout << "The result is: "
               << result.get_value()
               << std::endl;

    return 0;
}
```

Lecture 3 Outline

- Cilk++
- **Internally-Deterministic Algorithms**
- Priority-write Primitive
- Work Stealing Beyond Nested Parallelism
- Other Extensions
 - False Sharing
 - Work Stealing under Multiprogramming
- Emerging Memory Technologies

Nondeterminism



- Concerned about nondeterminism due to parallel scheduling orders and concurrency

Nondeterminism is problematic

- Debugging is painful
- Hard to reason about code
- Formal verification is hard
- Hard to measure performance



"Insanity: doing the same thing over and over again and expecting different results."
- Albert Einstein

Inherently Deterministic Problems

| | |
|------------------------|-------------------------|
| Breadth first search | Spanning forest |
| Suffix array | Minimum spanning forest |
| Remove duplicates | Maximal Independent set |
| Comparison sort | K-nearest neighbors |
| N-body | Triangle ray intersect |
| Delaunay triangulation | Delaunay refinement |

- **Wide coverage of real-world non-numeric problems**
- **Random numbers can be deterministic**

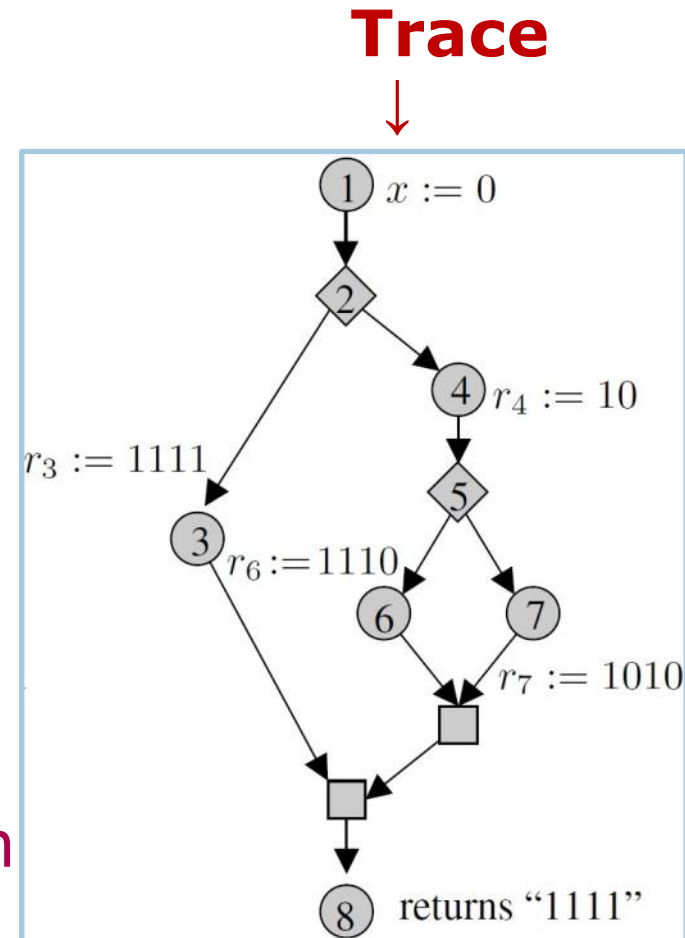
External vs. Internal Determinism

- **External: same input → same result**
- **Internal: same input → same intermediate states & same result**

Internal Determinism

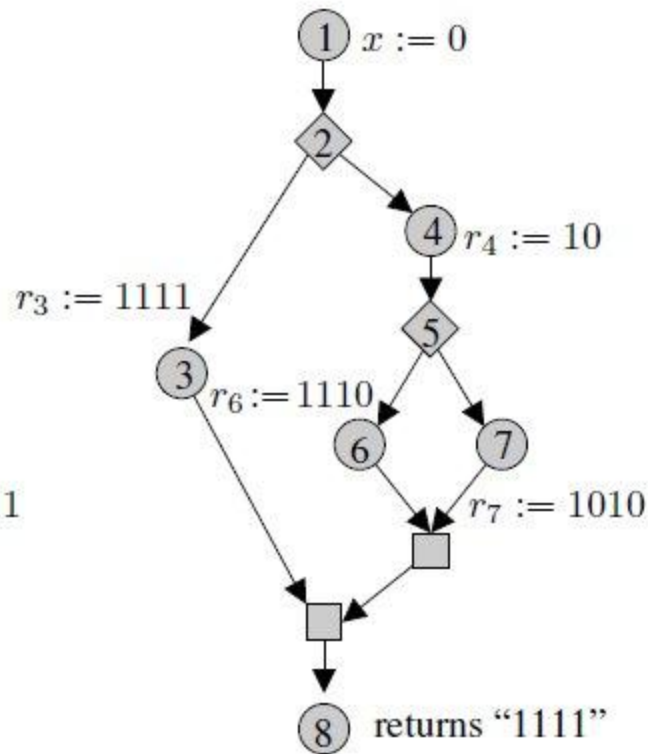
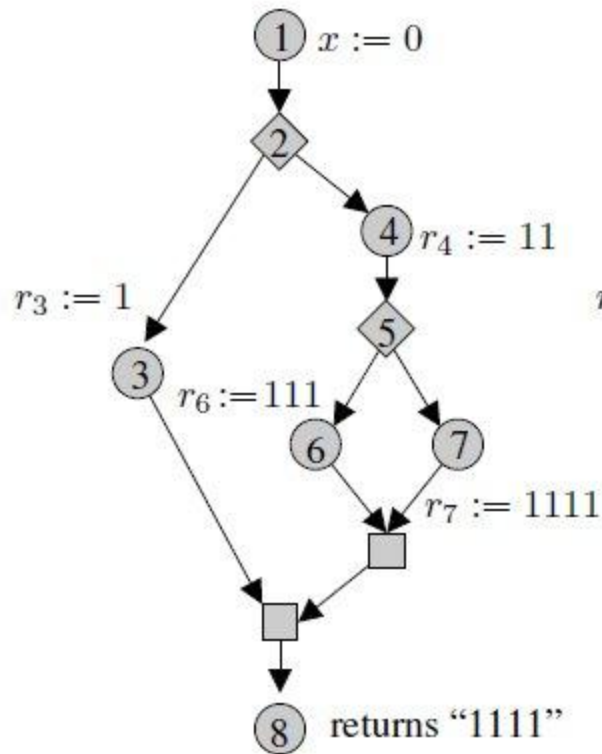
[Netzer, Miller '92]

- **Trace**: a computation's final state, intermediate states, and control-flow DAG
- **Internally deterministic**: for any fixed input, all possible executions result in equivalent traces (w.r.t. some level of abstraction)
 - Also implies external determinism
 - Provides sequential semantics



Internally deterministic?

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



Commutative + Nested Parallel → Internal Determinism

[Steele '90]

- **Commutativity**

- [Steele '90] define it in terms of memory operations
- [Cheng et al. '98] extend it to critical regions
- Two operations f and g commute if $f \circ g$ and $g \circ f$ have same final state and same return values

- **We look at commutativity in terms of arbitrary abstraction by introducing “commutative building blocks”**

- **We use commutativity strictly to get deterministic behavior, but there are other uses...**

System Approaches to Determinism

Determinism via

- **Hardware mechanisms** [Devietti et al. '11, Hower et al. '11]
- **Runtime systems and compilers** [Bergan et al. '10, Berger et al. '09, Olszewski et al. '09, Yu and Narayanasamy '09]
- **Operating systems** [Bergan et al. '10]
- **Programming languages/frameworks** [Bocchino et al. '09]

Commutative Building Blocks

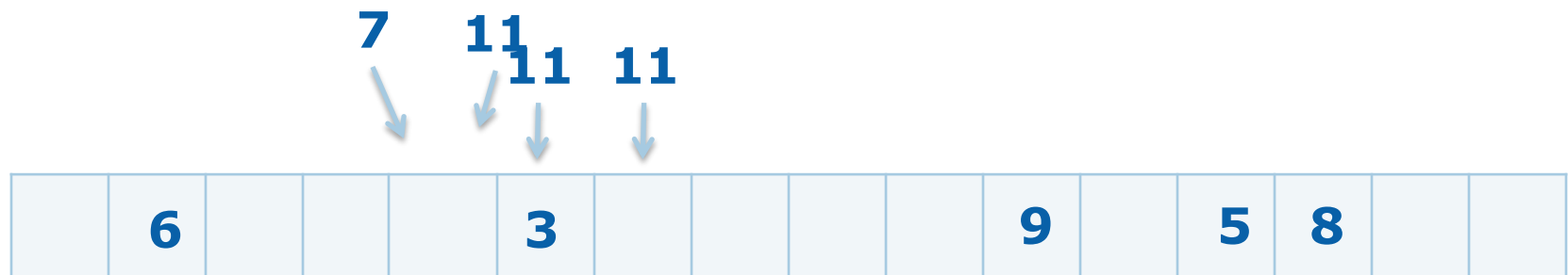
[Blelloch, Fineman, G, Shun '12]

- **Priority write**
 - pwrite, read
- **Priority reserve**
 - reserve, check, checkReset
- **Dynamic map**
 - insert, delete, elements
- **Disjoint set**
 - find, link
- **At this level of abstraction,
reads commute with reads &
updates commute with updates**

Dynamic Map

Using hashing:

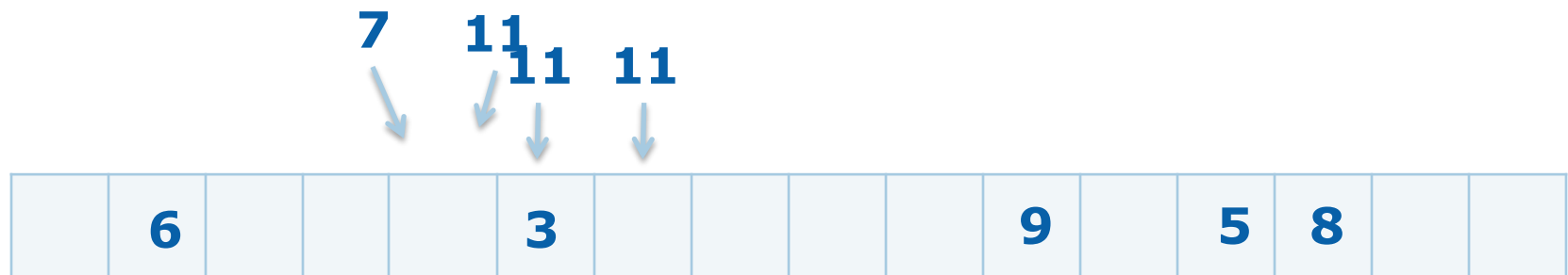
- Based on generic hash and comparison
- Problem: representation can depend on ordering. Also on which redundant element is kept.
- Solution: Use history independent hash table based on linear probing...once done inserting, representation is independent of order of insertion



Dynamic Map

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

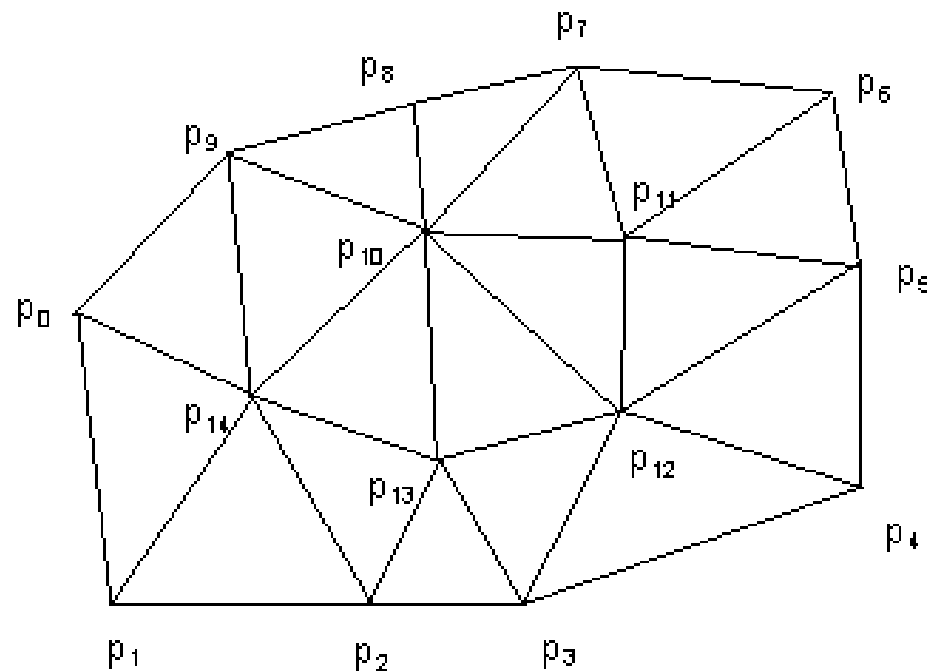


Internally Deterministic Problems

| Functional programming | History-independ. data structures | Deterministic reservations |
|------------------------|-----------------------------------|----------------------------|
| Suffix array | Remove duplicates | Spanning forest |
| Comparison sort | Delaunay refinement | Minimum spanning forest |
| N-body | | Maximal independent set |
| K-nearest neighbors | | Breadth first search |
| Triangle ray intersect | | Delaunay triangulation |
| | | Delaunay refinement |

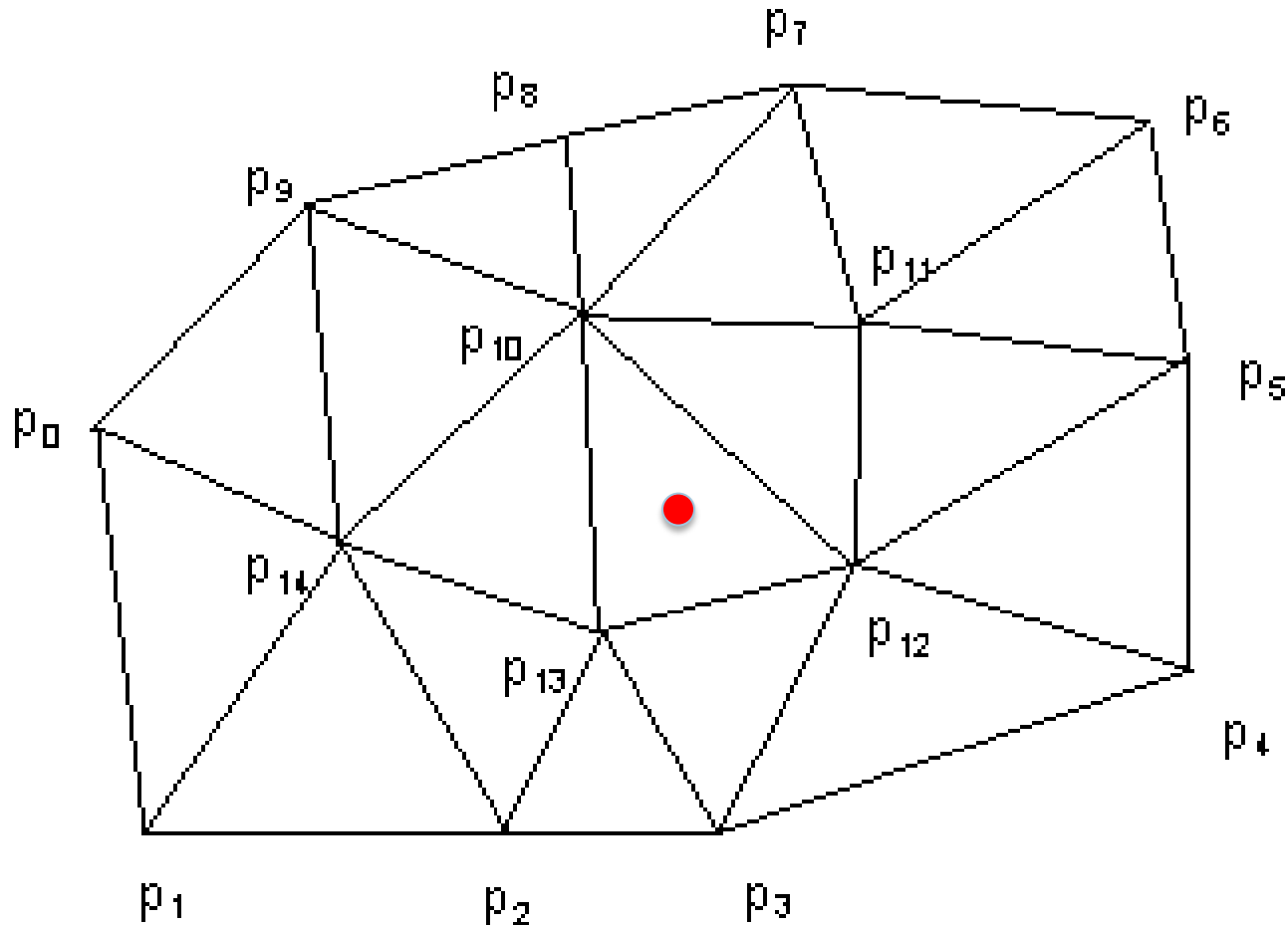
Delaunay Triangulation/Refinement

- **Incremental algorithm adds one point at a time, but points can be added in parallel if they don't interact**
- **The problem is that the output will depend on the order they are added.**



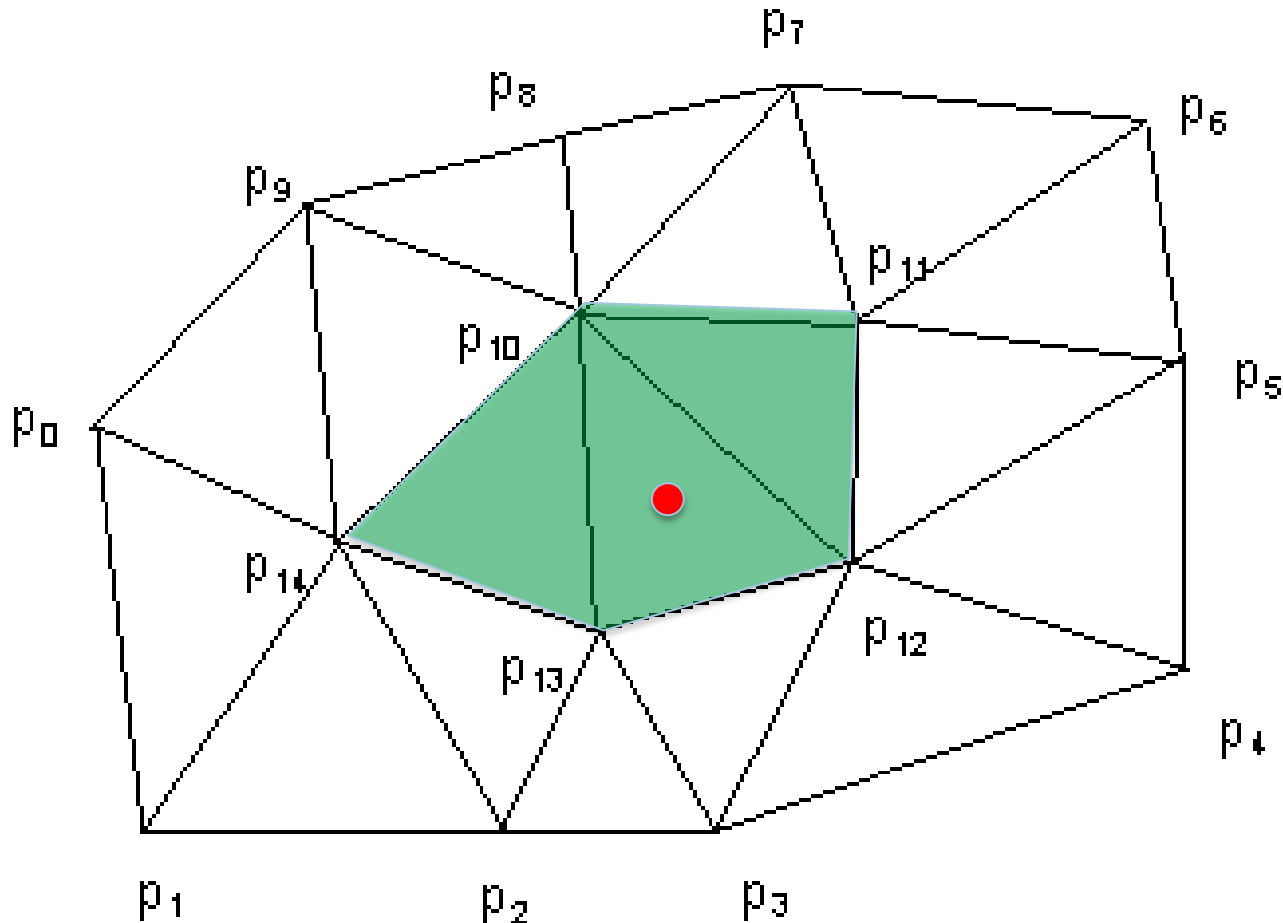
Delaunay Triangulation/Refinement

- Adding points deterministically



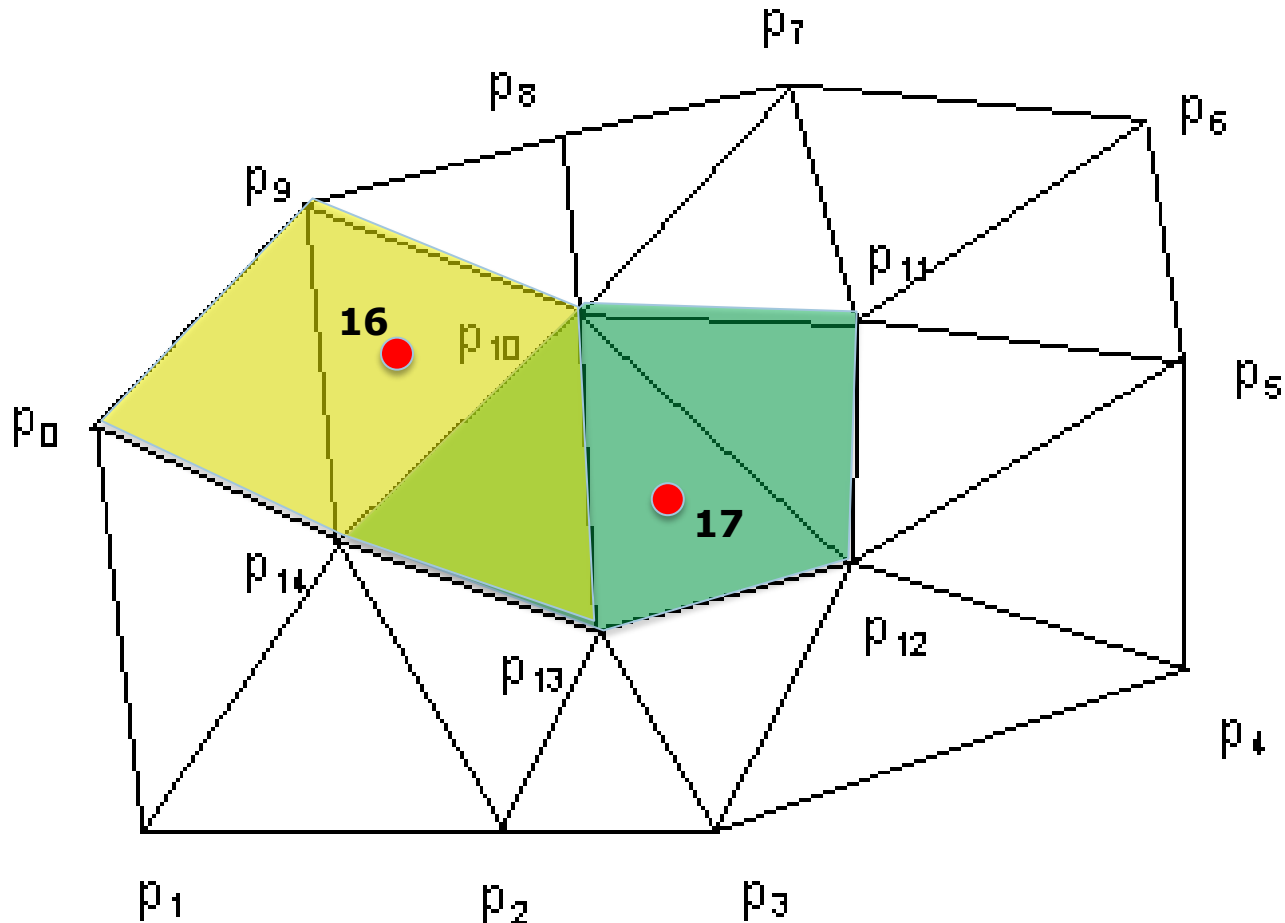
Delaunay Triangulation/Refinement

- Adding points deterministically



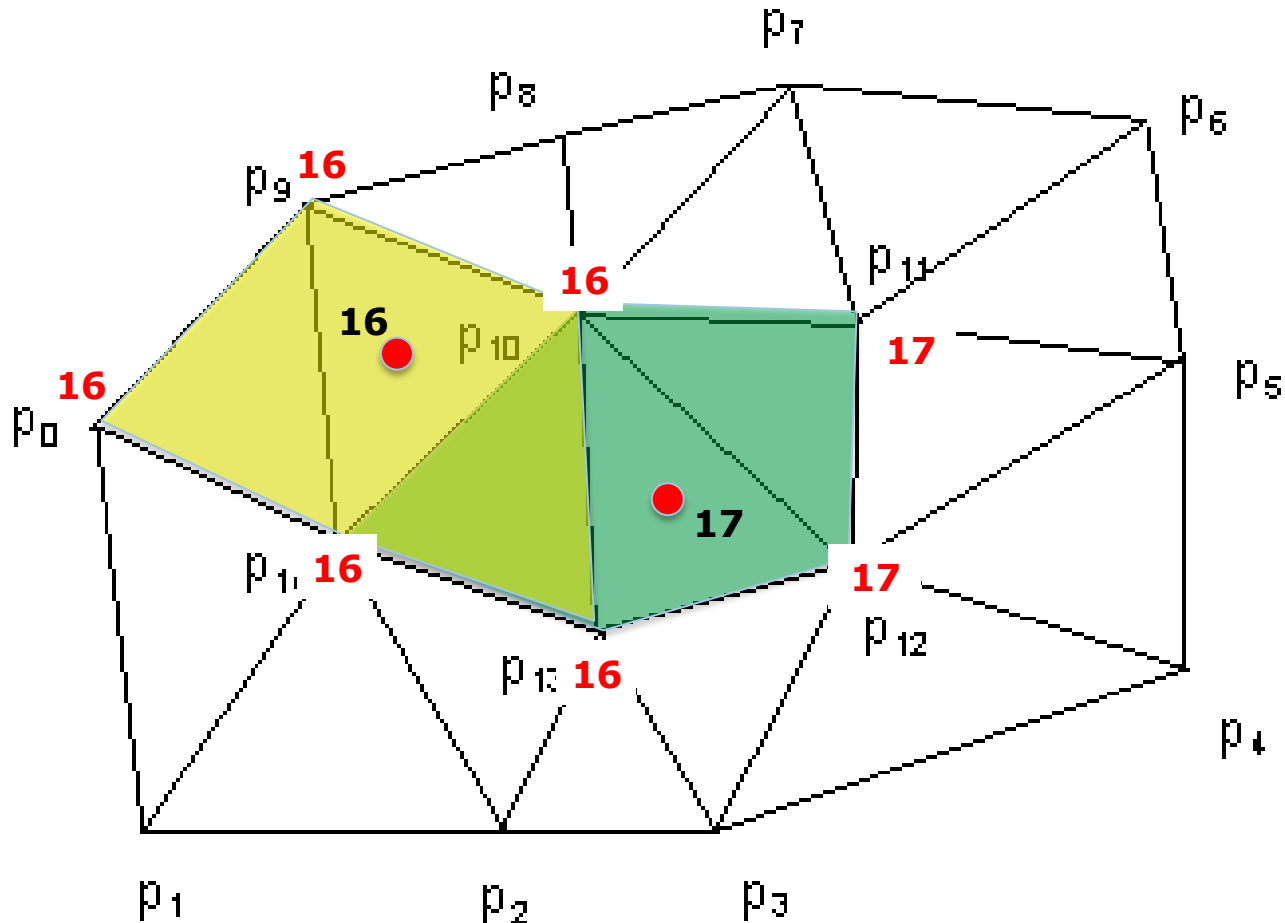
Delaunay Triangulation/Refinement

- Adding points deterministically



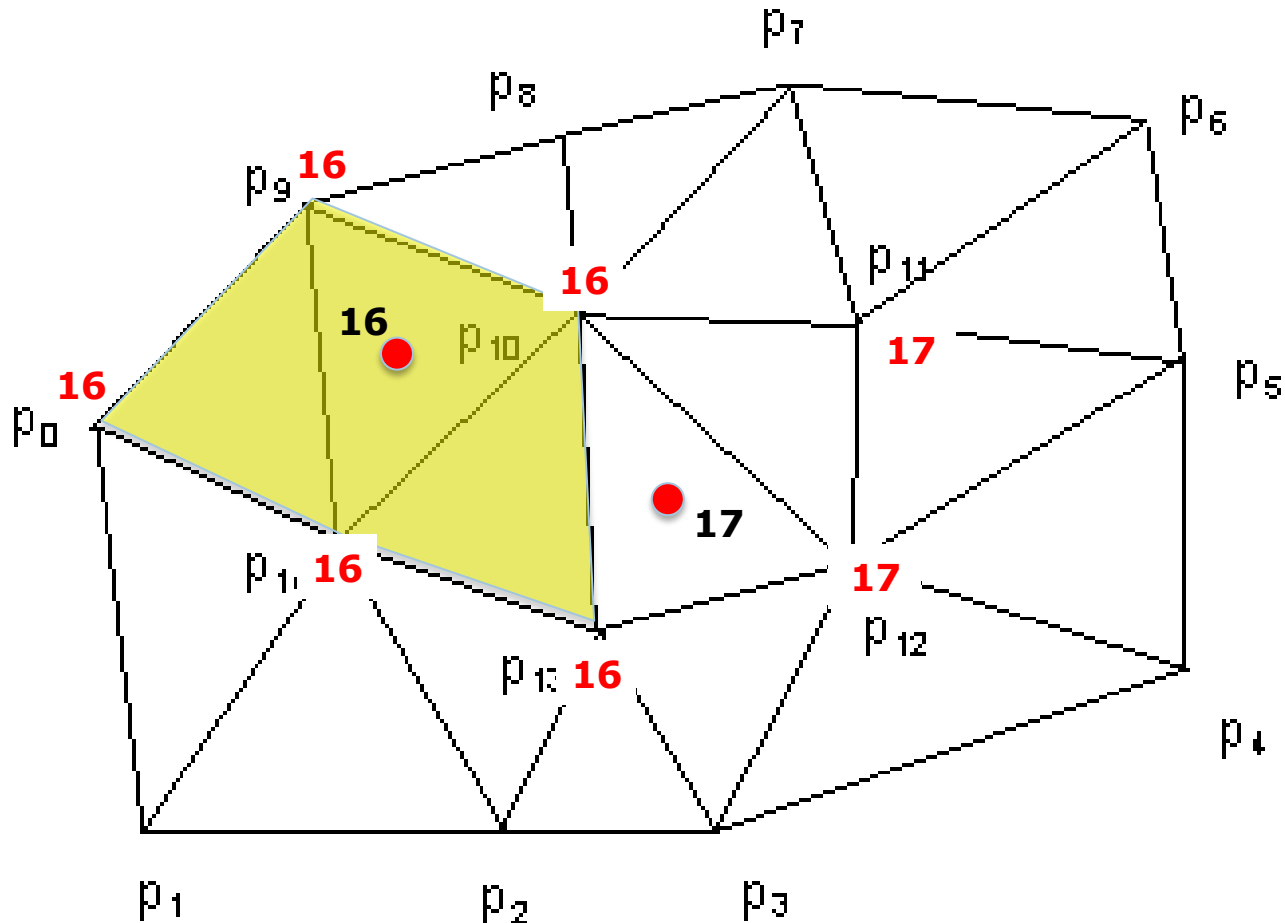
Delaunay Triangulation/Refinement

- Adding points deterministically



Delaunay Triangulation/Refinement

- Adding points deterministically



Deterministic Reservations

Generic framework

```
iterates = [1,...,n];  
while(iterates remain){  
  
    Phase 1: in parallel, all i in  
             iterates call reserve(i);  
  
    Phase 2: in parallel, all i in  
             iterates call commit(i);  
  
    Remove committed i's from  
    iterates;  
}
```

Note: Performance can be improved by processing prefixes of iterates in each round

Delaunay triangulation/refinement

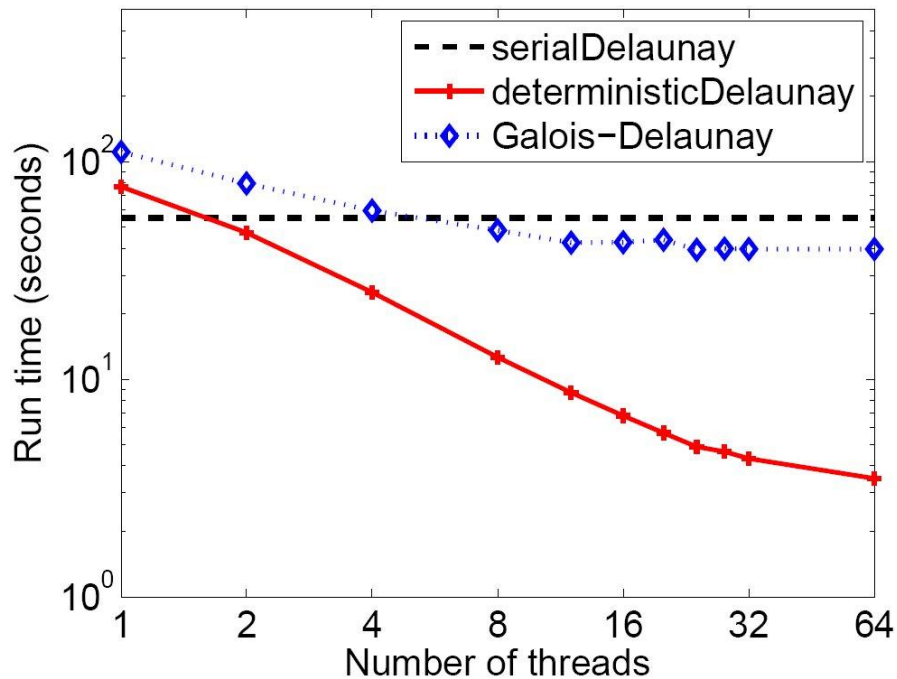
```
reserve(i){  
    find cavity;  
    reserve points in cavity;  
}  
  
commit(i){  
    check reservations;  
    if(all reservations successful){  
        add point and triangulate;  
    }  
}
```

Internally Deterministic Code

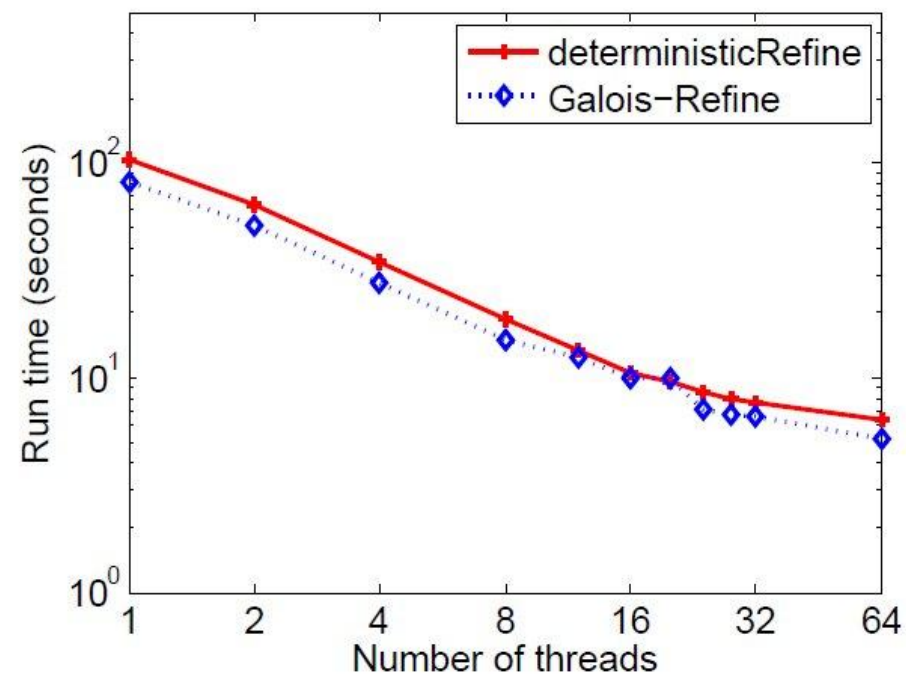
- **Implementations of benchmark problems**
 - Internally deterministic
 - Nondeterministic
 - Sequential
 - All require only 20-500 lines of code
- **Use nested data parallelism**
- **Used library of parallel operations on sequences: reduce, prefix sum, filter, etc.**

Experimental Results

Delaunay Triangulation



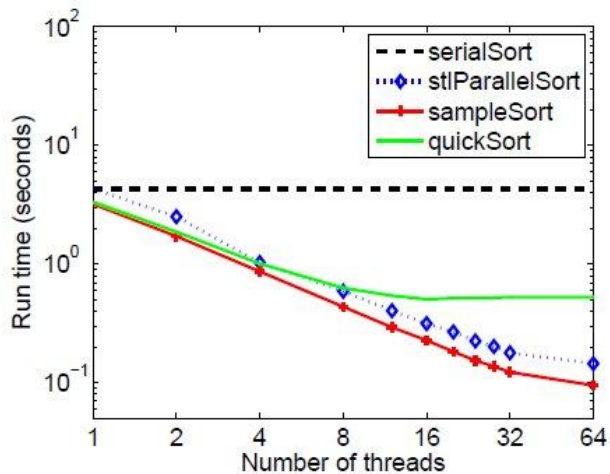
Delaunay Refinement



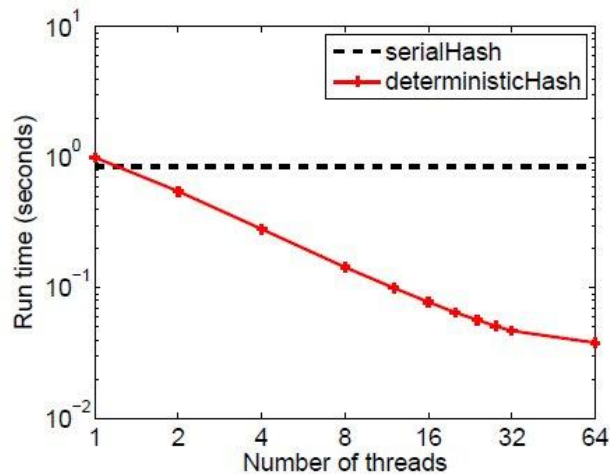
32-core Intel Xeon 7500 Multicore

Input Sets: 2M random points within a unit circle & 2M random 2D points from the Kuzmin distribution

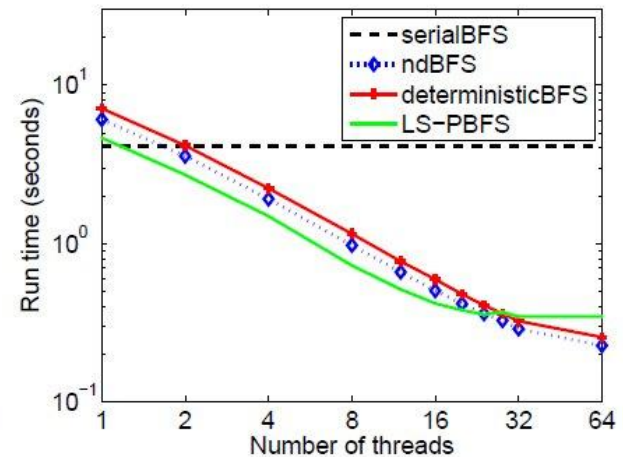
Experimental Results



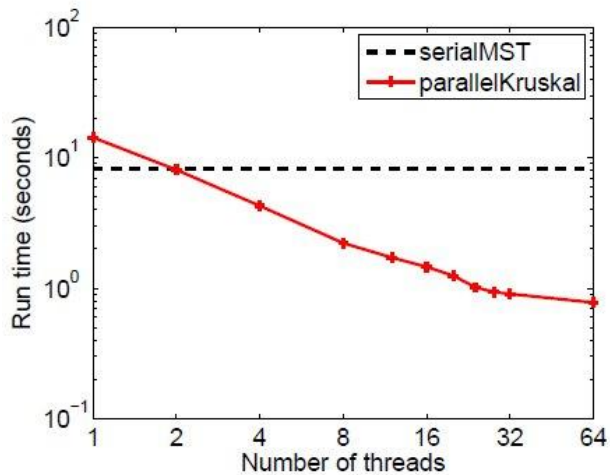
(a) comparison sorting algorithms with a **trigram** string of length 10^7



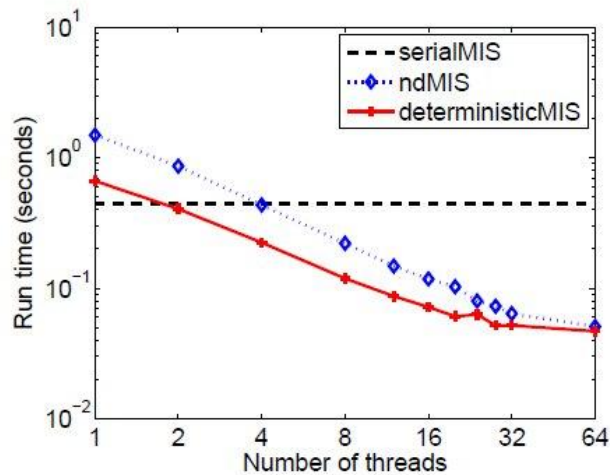
(b) remove duplicates algorithms with a **trigram** string of length 10^7



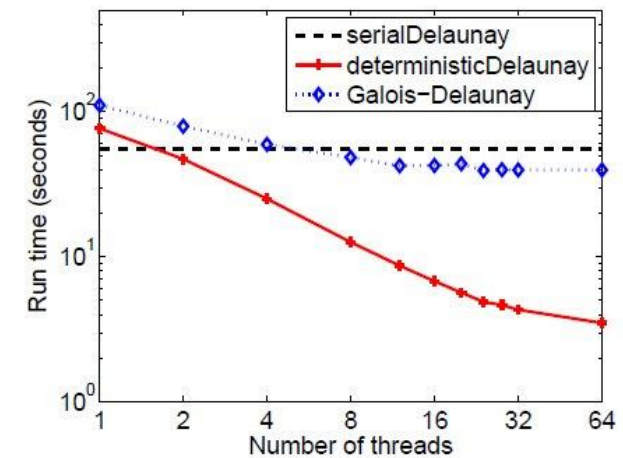
(c) BFS algorithms with a **random local graph** ($n = 10^7, m = 5 \times 10^7$)



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

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

Speedups on 40-core Xeon E7-8870

| Application Algorithm | 1 thread | 40 core | T_1/T_{40} | T_S/T_{40} |
|-----------------------------|-------------|------------|--------------|--------------|
| Integer Sort | | | | |
| serialRadixSort | 0.48 | – | – | – |
| parallelRadixSort | 0.299 | 0.013 | 23.0 | 36.9 |
| Comparison Sort | | | | |
| serialSort | 2.85 | – | – | – |
| sampleSort | 2.59 | 0.066 | 39.2 | 43.2 |
| Remove Duplicates | | | | |
| serialHash | 0.689 | – | – | – |
| parallelHash | 0.867 | 0.027 | 32.1 | 25.5 |
| Dictionary | | | | |
| serialHash | 0.574 | – | – | – |
| parallelHash | 0.748 | 0.025 | 29.9 | 23 |
| Breadth First Search | | | | |
| serialBFS | 2.61 | – | – | – |
| parallelBFS | 5.54 | 0.247 | 22.4 | 10.6 |
| Spanning Forest | | | | |
| serialSF | 1.733 | – | – | – |
| parallelSF | 5.12 | 0.254 | 20.1 | 6.81 |
| Min Spanning Forest | | | | |
| serialMSF | 7.04 | – | – | – |
| parallelKruskal | 14.9 | 0.626 | 23.8 | 11.2 |

| Application Algorithm | 1 thread | 40 core | T_1/T_{40} | T_S/T_{40} |
|-------------------------------|-------------|------------|--------------|--------------|
| Maximal Ind. Set | | | | |
| serialMIS | 0.405 | – | – | – |
| parallelMIS | 0.733 | 0.047 | 14.1 | 8.27 |
| Maximal Matching | | | | |
| serialMatching | 0.84 | – | – | – |
| parallelMatching | 2.02 | 0.108 | 18.7 | 7.78 |
| K-Nearest Neighbors | | | | |
| ocfTreeNeighbors | 24.9 | 1.16 | 21.5 | – |
| Delaunay Triangulation | | | | |
| serialDelaunay | 56.3 | – | – | – |
| parallelDelaunay | 76.6 | 2.6 | 29.5 | 21.7 |
| Convex Hull | | | | |
| serialHull | 1.01 | – | – | – |
| quickHull | 1.655 | 0.093 | 17.8 | 10.9 |
| Suffix Array | | | | |
| serialKS | 17.3 | – | – | – |
| parallelKS | 11.7 | 0.57 | 20.5 | 30.4 |
| Ray Casting | | | | |
| kdTree | 7.32 | 0.334 | 21.9 | – |

Problem Based Benchmark Suite

<http://www.cs.cmu.edu/~pbbs/>

Goal: A set of “problem based benchmarks”

Must satisfy a particular input-output interface,
but there are no rules on the techniques used

Measure the quality of solutions based on:

- **Performance and speedup** over a variety of input types and w.r.t. best sequential implementations
- **Quality of output.** Some benchmarks don't have a right answer or are approximations
- **Complexity of code.** Lines of code & other measures
- **Determinism.** The code should always return the same output on same input
- **Generic.** Code should be generic over types
- **Correctness guarantees**
- **Easily analyze performance,** at least approximately

Lecture 3 Outline

- Cilk++
- Internally-Deterministic Algorithms
- **Priority-write Primitive**
- Work Stealing Beyond Nested Parallelism
- Other Extensions
 - False Sharing
 - Work Stealing under Multiprogramming
- Emerging Memory Technologies

Priority Write as a Parallel Primitive

[Shun, Blelloch, Fineman, G]

- **Priority-write**: when there are multiple writes to a location, possibly concurrently, the value with the highest priority is written
 - E.g., write-with-min: for each location, min value written wins (used earlier in Delaunay Refinement)

A := 5 B := 17 B := 12 A := 9 A := 8

yields A = 5 and B = 12

- **Useful parallel primitive**:
 - + Low contention even under high degrees of sharing
 - + Avoids many concurrency bugs since commutes
 - + Useful for many algorithms & data structures

Priority-Write Performance

Times for 5 runs of 100 million operations to x random locations
on 40-core Intel machine

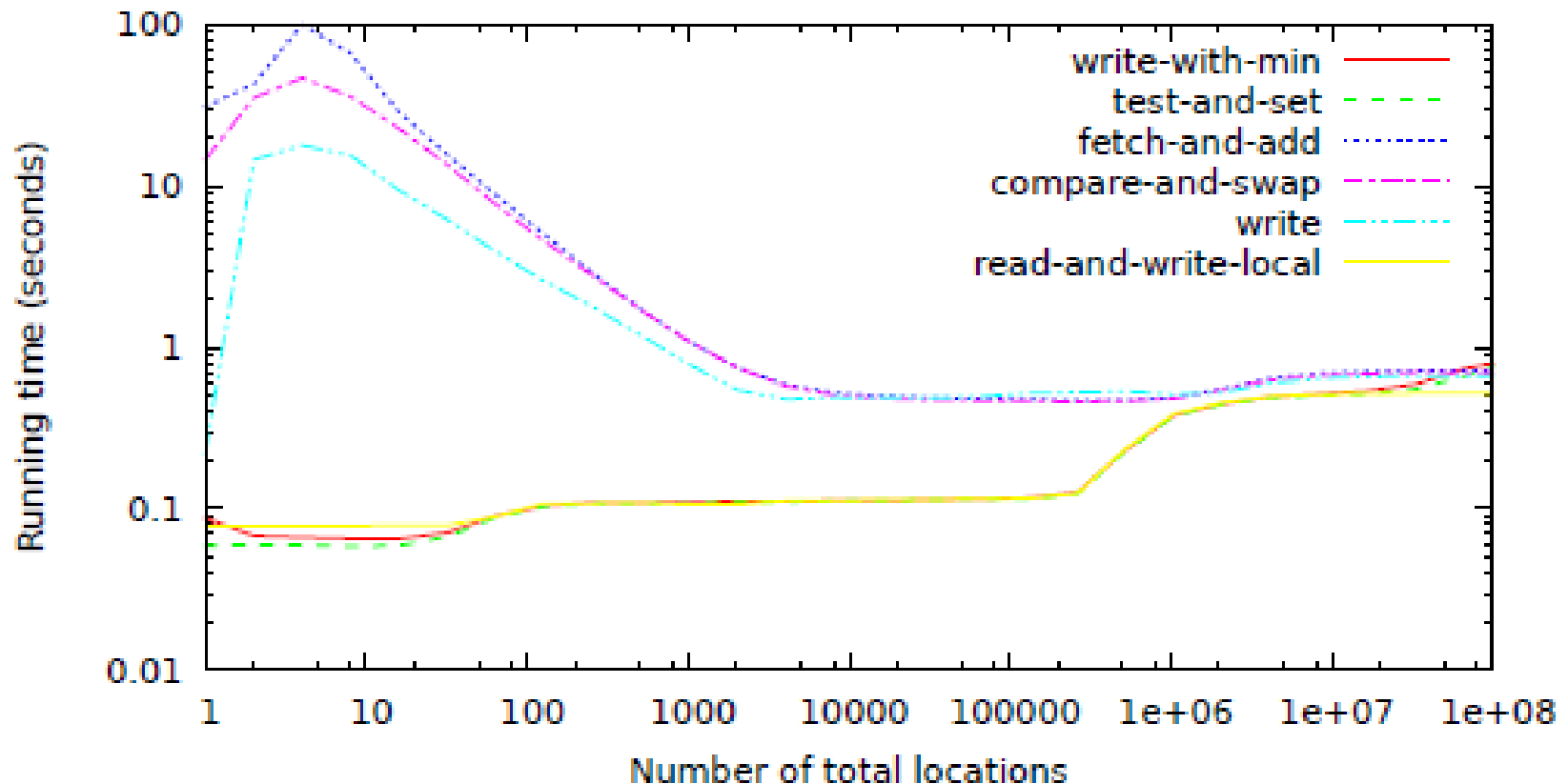


Figure 1. Time for six different operations types on a 40-core Intel Nehalem under various degrees of sharing (log-log scale).

© Similar results on 48-core AMD Opteron 6168



Theoretical Justification

Lemma: Consider a collection of n distinct priority-write operations to a single location, where at most p randomly selected operations occur concurrently at any time. Then the number of CAS attempts is $O(p \ln n)$ with high probability.

Idea: Let X_k be an indicator for the event that the k th priority-write performs an update. Then $X_k = 1$ with probability $1/k$, as it updates only if it is the highest-priority of all k earliest writes. The expected number of updates is then given by $E[X_1 + \dots + X_n] = 1/1 + 1/2 + 1/3 + \dots + 1/n = H_n$.

Priority-Write in Algorithms

- **Take the maximum/minimum of set of values**
- **Avoiding nondeterminism since commutative**
- **Guarantee progress in algorithm:
highest priority thread will always succeed**
- **Deterministic Reservations: speculative parallel
FOR loop (use iteration as priority)**

Priority Writes in Algorithms

- **Parallel version of Kruskal's minimum spanning-tree algorithm so that the minimum-weight edge into a vertex is always selected**
- **Boruvka's algorithm to select the minimum-weight edge**
- **Bellman-Ford shortest paths to update the neighbors of a vertex with the potentially shorter path**
- **Deterministic Breadth-First Search Tree**

E.g., Breadth-First Search Tree

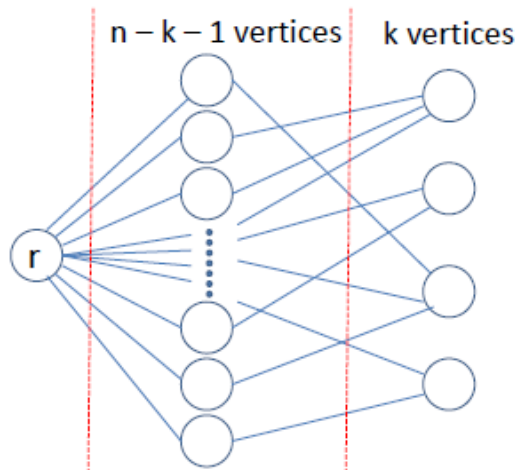
Frontier = {source vertex}

In each round:

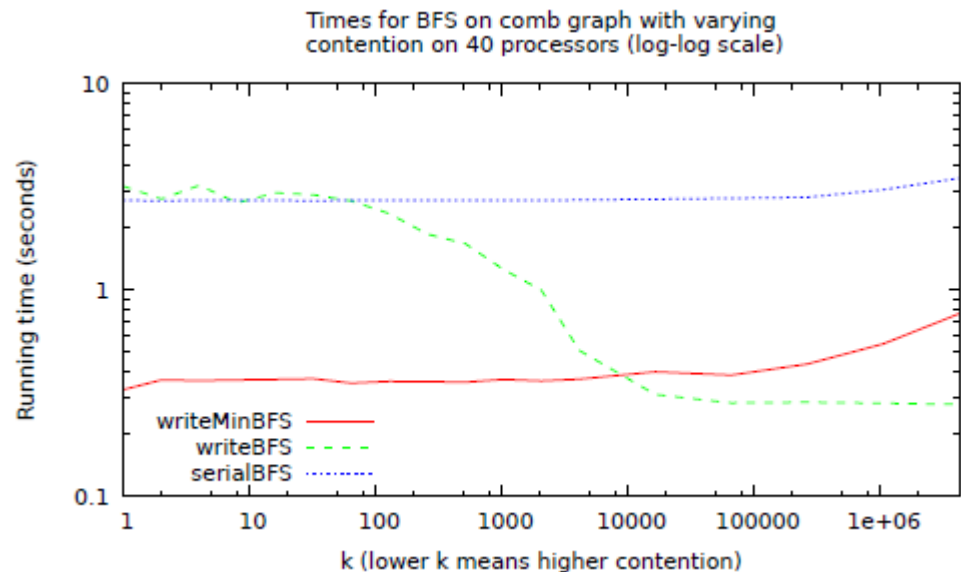
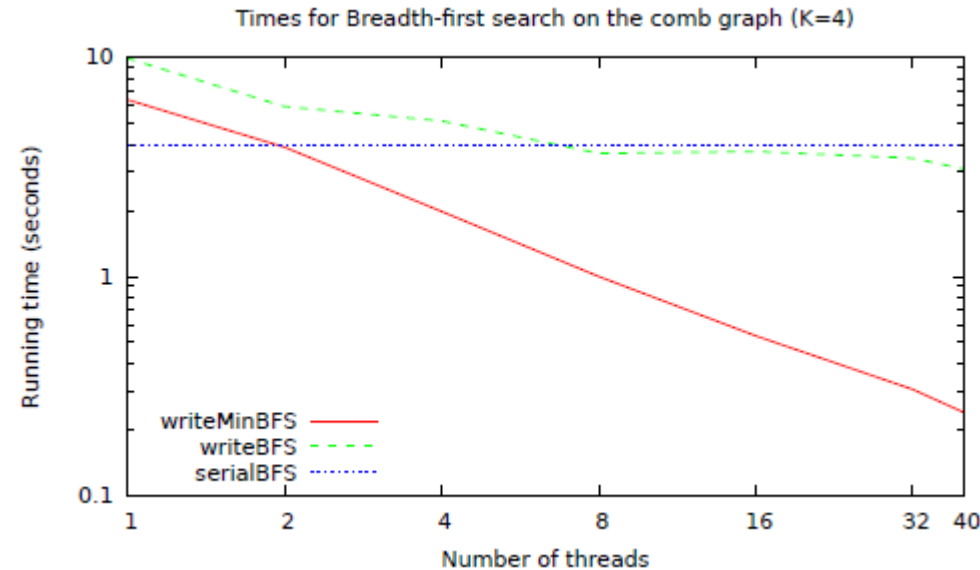
In parallel for all v in Frontier

Remove v ;

Attempt to place all v 's
neighbors in Frontier;



Input: Comb Graph



Priority-Write Definition

```
procedure PRIORITYWRITE(addr, newval, comp)  
    oldval  $\leftarrow$  *addr  
    while comp(newval, oldval) do  
        if CAS(addr, oldval, newval) then  
            return  
        else  
            oldval  $\leftarrow$  *addr  
        end if  
    end while  
end procedure
```

Priority-Writes on Locations

- **Efficient implementation of a more general dictionary-based priority-write where the writes/inserts are made based on keys.**
 - E.g., all writers might insert a character string into a dictionary with an associated priority
 - Use for prioritized remove-duplicates algorithm

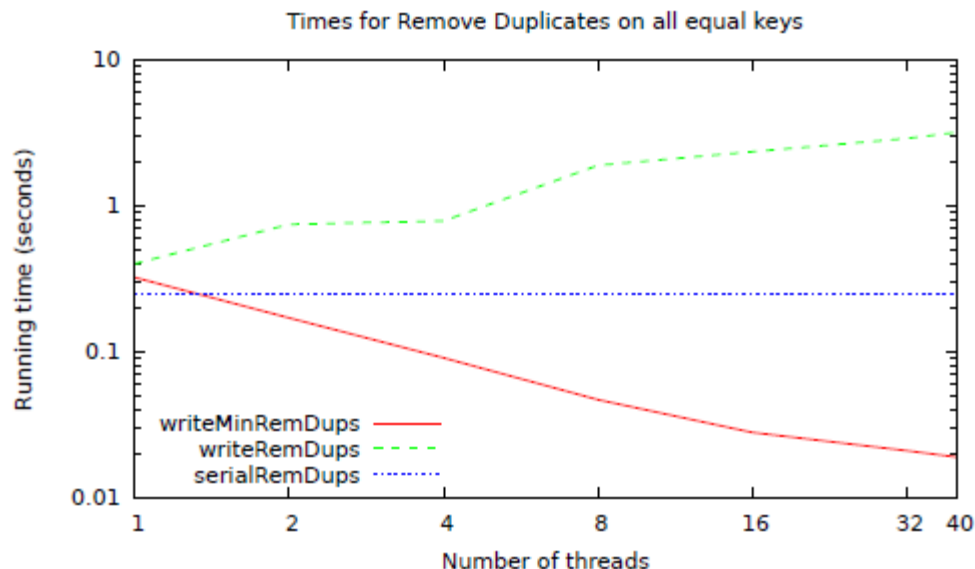


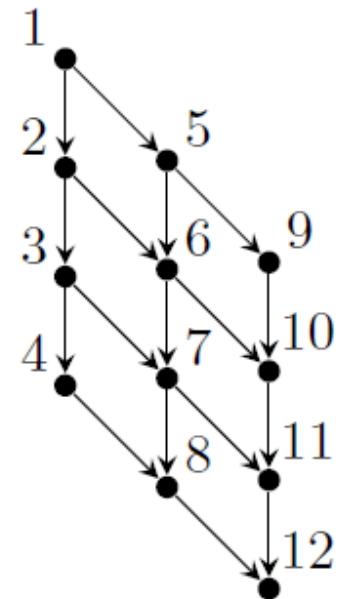
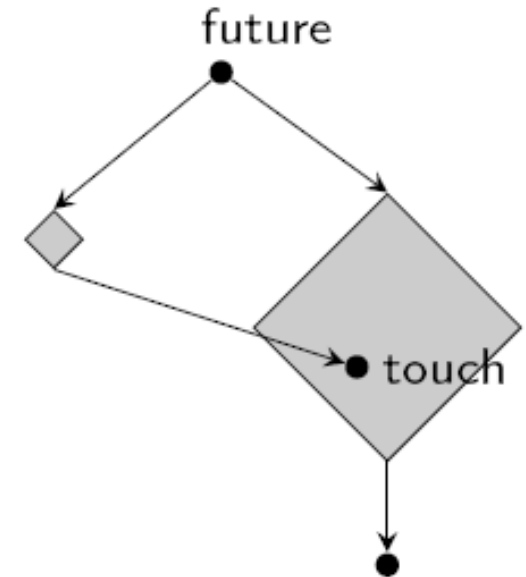
Figure 9. Remove Duplicates times on the allEqual sequence (log-log scale)

Lecture 3 Outline

- Cilk++
- Internally-Deterministic Algorithms
- Priority-write Primitive
- **Work Stealing Beyond Nested Parallelism**
- Other Extensions
 - False Sharing
 - Work Stealing under Multiprogramming
- Emerging Memory Technologies

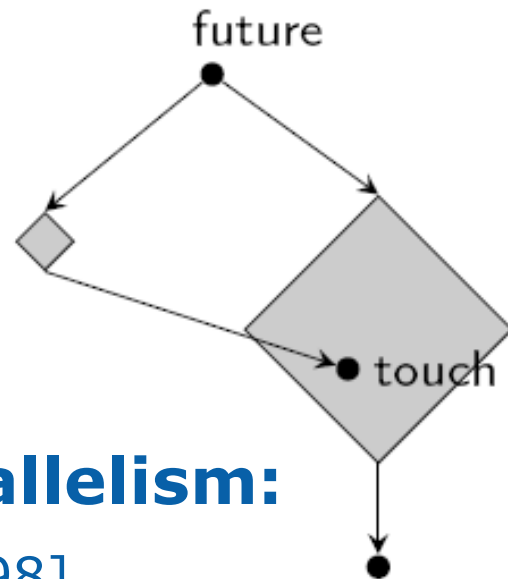
Parallel Futures

- Futures [Halstead '85], in Multilisp
 - Parallelism no longer nested
 - Here: explicit future and touch keywords
 - E.g. Halstead's quicksort, pipelining tree merge [Blelloch, Reid-Miller '97]
- Strictly more expressive than fork/join
 - E.g. can express parallel pipelining
- ... but still deterministic!



Work Stealing for Futures?

- **Implementation choices:**
 - **What to do when touch causes processor to stall?**
- **Previous work beyond nested parallelism:**
 - **Bound # of steals for WS** [Arora et al. '98]
 - ***We show: not sufficient to bound WS overhead, once add futures!***



Summary of previous work

Nested Parallelism:

$O(Pd)$ steals, Overheads additive in # of steals

Beyond Nested Parallelism:

$O(Pd)$ steals, # steals can't bound overheads

Bounds for Work Stealing with Futures

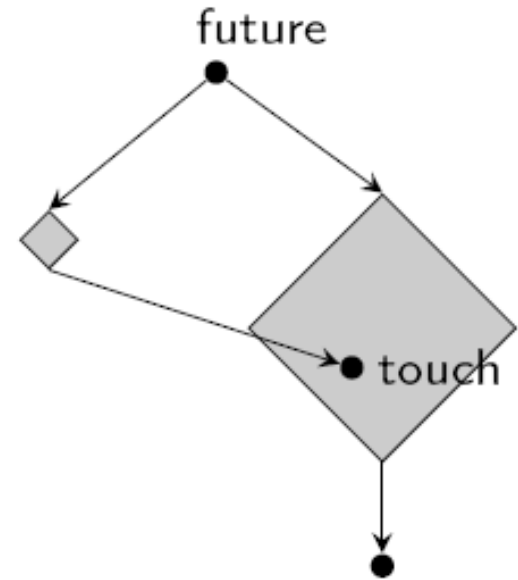
[Spoonhower, Blelloch, G, Harper '09]

Extend study of Work Stealing (WS) to Futures:

- Study “deviations” as a replacement for “steals”
 - Classification of deviations arising with futures
 - Tight bounds on WS overheads as function of # of deviations
- Give tight upper & lower bounds on # of deviations for WS
 - $\Theta(P_d + T_d)$, where T is # of touches
- Characterize a class of programs using futures effectively
 - Only $O(P_d)$ deviations

Futures + Parallelism

- Processor can **stall** when:
 1. No more tasks in local work queue
 2. Current task is waiting for a value computed by another processor
- Existing WS only **steals** in case 1
 - We call these ***parsimonious*** schedulers (i.e., pays the cost of a steal only when it must)
- Thus, in case 2, stalled processor jumps to other work on its local work queue



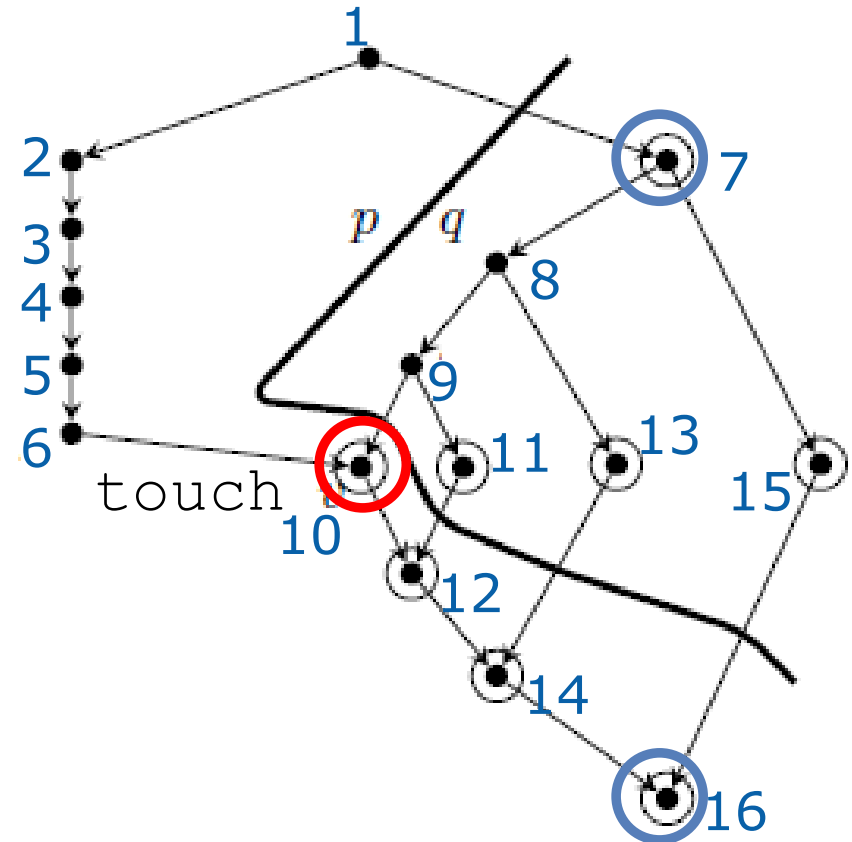
Deviations

A **deviation** (from the sequential schedule) occurs when...

- a processor p visits a node n ,
 - the sequential schedule visits n' immediately before n
 - ...but p did not.
- **Used by** [Acar, Blelloch, Blumofe '02] **to bound additional cache misses in nested parallelism**
 - **Our work: use deviations as means to bound several measures of performance**
 - Bound # of “slow clone” invocations (\approx computation overhead)
 - Bound # of cache misses in private LRU cache

Sources of Deviations

- In nested parallelism:
 - at steals & joins
 - # deviations $\leq 2 \times$ # steals
- With futures:
 - at steals & joins ○
 - at touches ○
 - indirectly after touches (rest)



Bounding WS Overheads

Δ = # of deviations

Invocations of slow clones

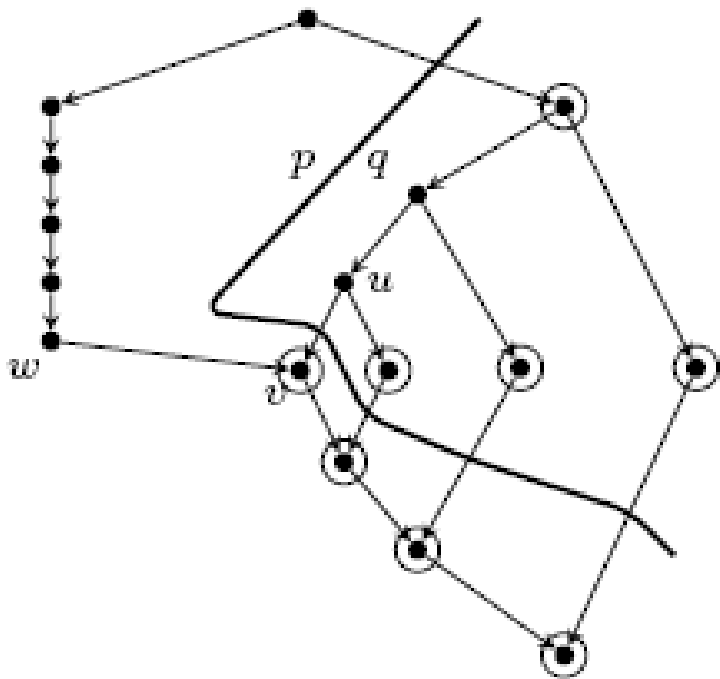
- Theorem: # of slow clone invocations $\leq \Delta$
- Lower bound: # of slow clone invocations is $\Omega(\Delta)$

Cache misses (extension of [Acar, Blelloch, Blumofe '02])

- Theorem: # of cache misses $< Q_1(M) + M \Delta$
 - Each processor has own LRU cache; under dag consistency
 - M = size of a (private) cache
 - $Q_1(M)$ = # of cache misses in sequential execution

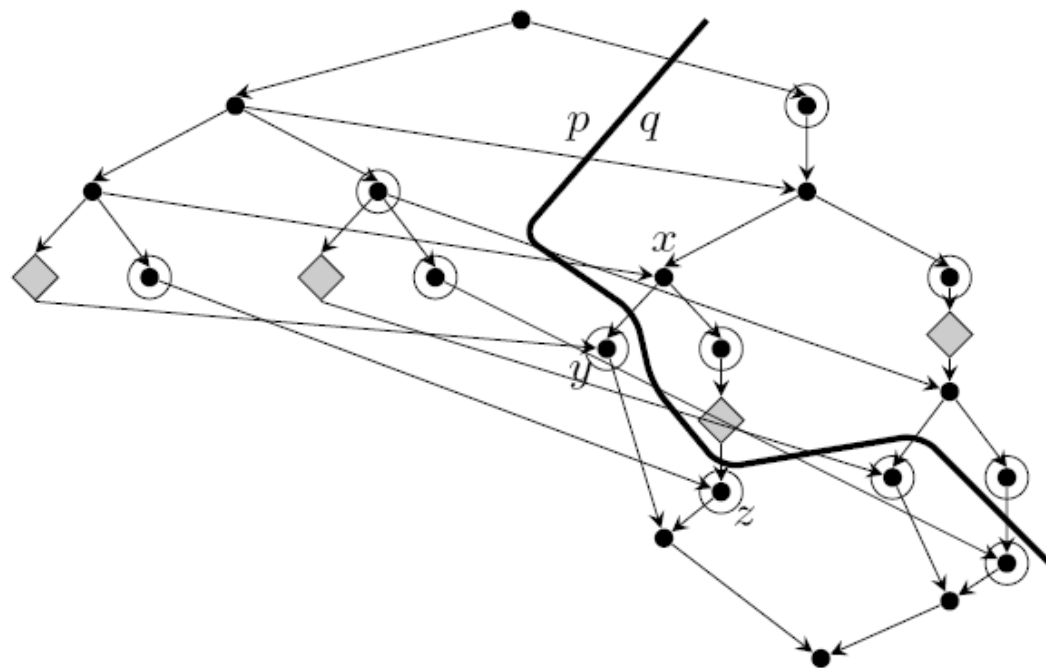
Deviations: Example Graphs

2 processors: p & q



1 future, 1 touch,
1 steal, span=d

$\Omega(d)$ deviations



T futures, T touches,
1 steal, $O(\log T)$ span

$\Omega(T)$ deviations

Bounding Deviations, Upper Bound

Main Theorem:

∀ computations derived from futures with depth d and T touches, the expected # deviations by any parsimonious WS scheduler on P processors is $O(Pd + Td)$

- First term $O(Pd)$ based on previous bound on # of steals
- Second term $O(Td)$ from indirect deviations after touches

Proof relies on:

- Structure of graphs derived from uses of futures
- Behavior of parsimonious WS

Pure Linear Pipelining

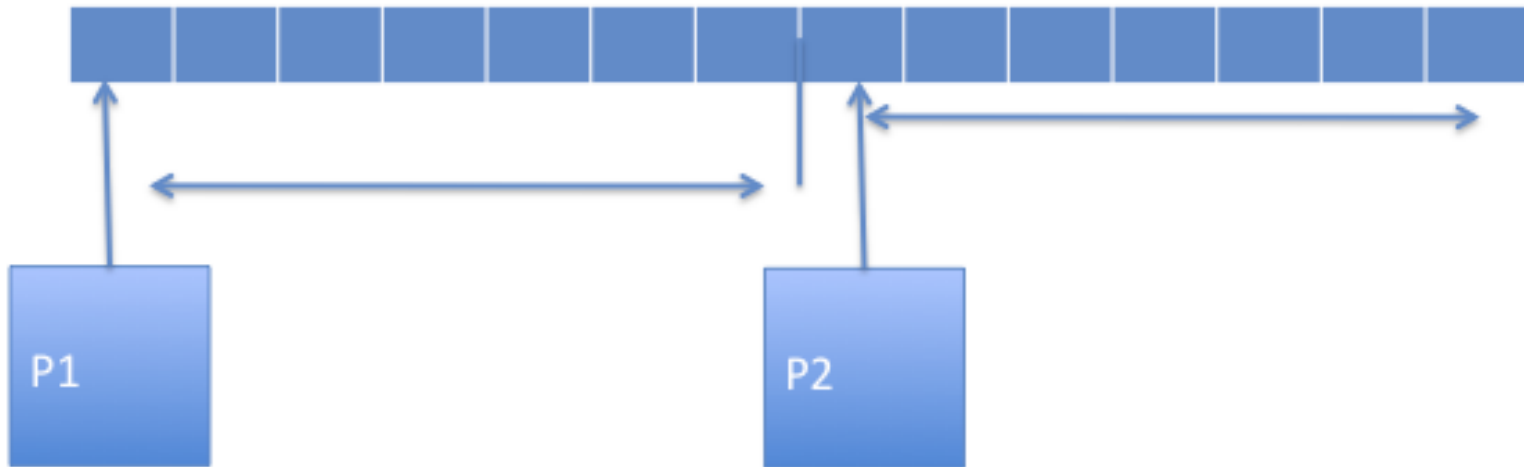
- Identified restricted use case w/ less overhead
 - # of deviations is $O(Pd)$
- Includes producer-consumer examples with streams, lists, one-dimensional arrays

Lecture 3 Outline

- Cilk++
- Internally-Deterministic Algorithms
- Priority-write Primitive
- Work Stealing Beyond Nested Parallelism
- **Other Extensions**
 - False Sharing
 - Work Stealing under Multiprogramming
- Emerging Memory Technologies

False Sharing

Block of size B shared by P1 and P2



False Sharing: $B/2$ cache misses incurred by P1 and by P2

Block-Resilience

[Cole, Ramachandran '12]

- **Hierarchical Balanced Parallel** (HBP) computations use balanced fork-join trees and build richer computations through sequencing and recursion
- Design HBP with good sequential cache complexity, and good parallelism
- Incorporate **block resilience** in the algorithm to guarantee low overhead due to false sharing
- Design **resource-oblivious algorithms** (i.e., with no machine parameters in the algorithms) that are analyzed to perform well (across different schedulers) as a function of the number of parallel tasks generated by the scheduler

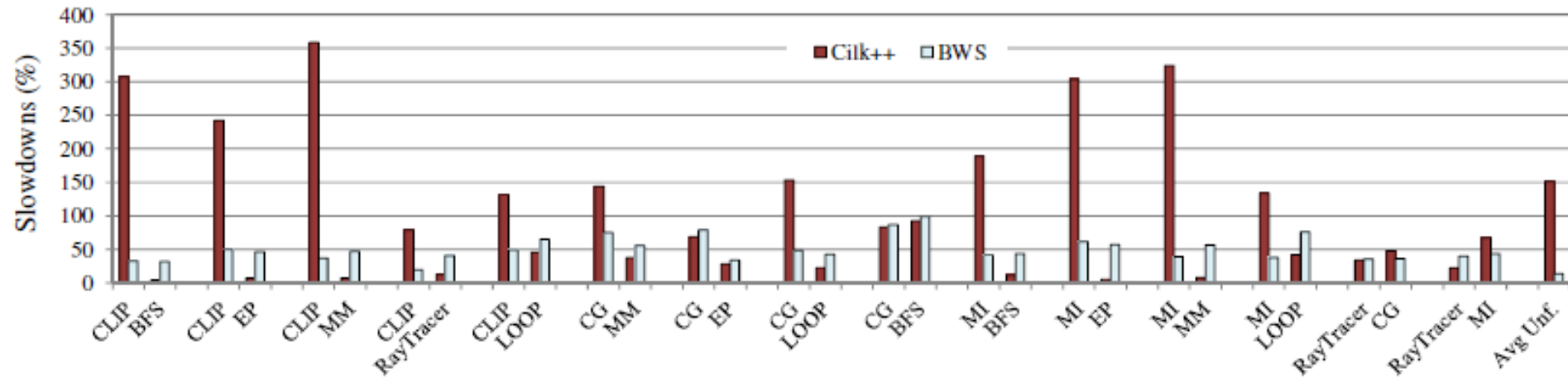
BOUNDS FOR RANDOMIZED WORK STEALING (RWS)

| Block Resilient HBP Algorithm | RWS Expected # Steals, S with FS Misses [Cole-R12c] | Cache Misses with S Steals [Cole-R12a] | FS Misses [Cole-R12b] |
|----------------------------------|--|--|--------------------------|
| Scans, MT | $p \cdot (\log n + \frac{b}{s}B)$ | $Q + S$ [FS06,CR12a] | $S \cdot B$ |
| RM to BI | $p \cdot (\log n + \frac{b}{s}B)$ | $Q + S \cdot B$ | $S \cdot B$ |
| MM, Strassen | $p \cdot (\log^2 n + \frac{b}{s}B \log n)$ | $Q + S^{\frac{1}{3}} \frac{n^2}{B} + S$ | $S \cdot B$ |
| Depth-n-MM | $p \cdot (n + \frac{b}{s}n\sqrt{B})$ | $Q + S^{\frac{1}{3}} \frac{n^2}{B} + S$ [FS06,CR12a] | $S \cdot B$ |
| I-GEP | $p \cdot (n \cdot \log^2 n + \frac{b}{s}n\sqrt{B})$ | $Q + S^{\frac{1}{3}} \frac{n^2}{B} + S$ [FS06,CR12a] | $S \cdot B$ |
| BI to RM for MM and FFT | $p \cdot (\log n + \frac{b}{s}B)$ | $Q + S \cdot B + \frac{n^2}{B} \log \log_B n$ | $S \cdot B$ |
| LCS | $p(1 + \frac{b}{s}) \cdot n^{\log_2 3}$ | $Q + n\sqrt{S}/B + S$ [FS06,CR12a] | $S \cdot B$ |
| FFT, sort | $p \cdot (\log n \cdot \log \log n + \frac{b}{s}B \log_B n)$ | $C_{\text{sort}} = O(Q + S \cdot B + \frac{n}{B} \frac{\log n}{\log[(n \log n)/S]})$ | $S \cdot B$ |
| List Ranking | $p \cdot \log n \cdot \log \log n \cdot (\log n + \frac{b}{s}B)$ | $Q + C_{\text{sort}} \cdot \log n$ | $S \cdot B$ |

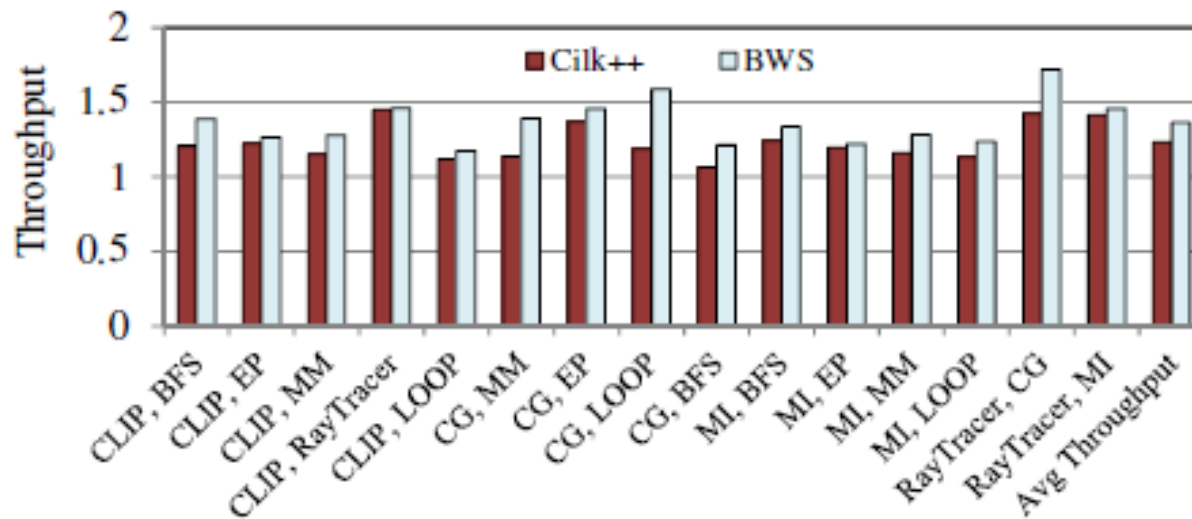
Multiple Work-Stealing Schedulers at Once?

- **Dealing with multi-tenancy**
- **Want to run at same time**
- **Schedulers must provide throughput + fairness**
 - Failed steal attempts not useful work
 - Yielding at failed steal attempts leads to unfairness
 - BWS [Ding et al. '12] decreases average unfairness from 124% to 20% and increases throughput by 12%
- **Open: What bounds can be proved?**

Unfairness



Throughput

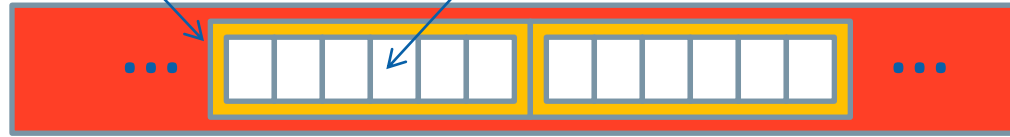


Lecture 3 Outline

- **Cilk++**
- **Internally-Deterministic Algorithms**
- **Priority-write Primitive**
- **Work Stealing Beyond Nested Parallelism**
- **Other Extensions**
 - False Sharing
 - Work Stealing under Multiprogramming
- **Emerging Memory Technologies**

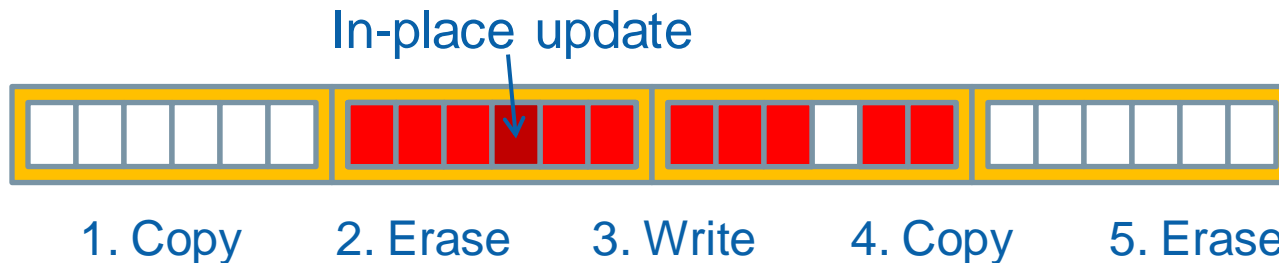
NAND Flash Chip Properties

Block (64-128 pages) Page (512-2048 B)

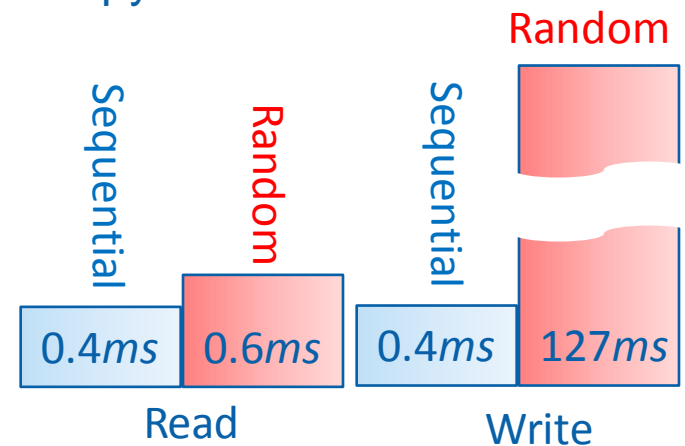


Read/write pages,
erase blocks

- Write page once after a block is erased



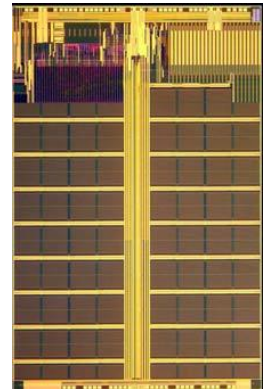
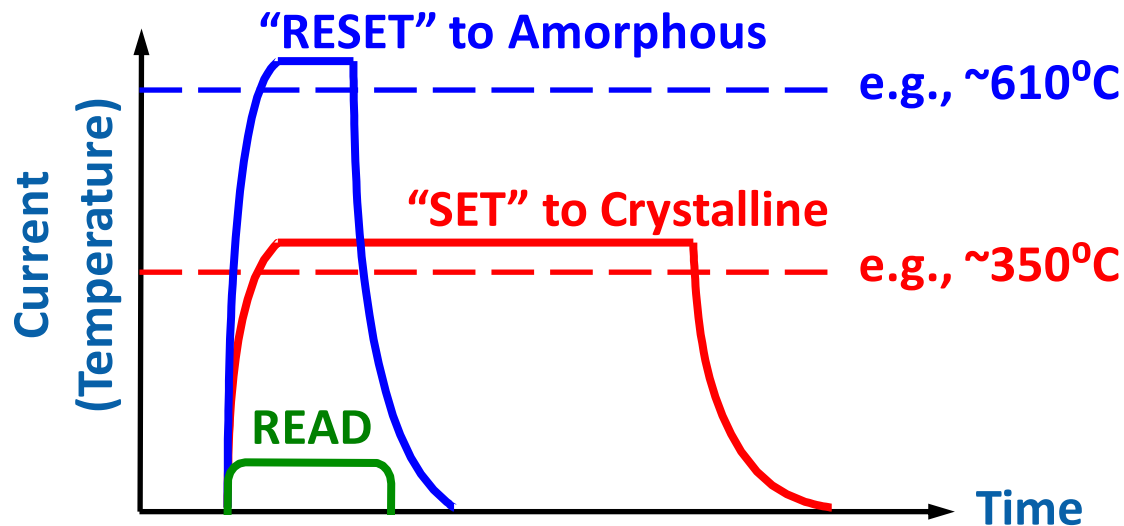
- Expensive operations:
 - In-place updates
 - Random writes



These quirks are now hidden by Flash/SSD firmware

Phase Change Memory (PCM)

- **Byte-addressable non-volatile memory**
- **Two states of phase change material:**
 - Amorphous: high resistance, representing "0"
 - Crystalline: low resistance, representing "1"
- **Operations:**



Comparison of Technologies

| | DRAM | PCM | NAND Flash |
|--------------------|------------------|------------------------|----------------------|
| Page size | 64B | 64B | 4KB |
| Page read latency | 20-50ns | ~ 50ns | ~ 25 μ s |
| Page write latency | 20-50ns | ~ 1 μ s | ~ 500 μ s |
| Write bandwidth | ~GB/s per die | 50-100 MB/s per die | 5-40 MB/s per die |
| Erase latency | N/A | N/A | ~ 2 ms |
| Endurance | ∞ | $10^6 - 10^8$ | $10^4 - 10^5$ |
| Read energy | 0.8 J/GB | 1 J/GB | 1.5 J/GB [28] |
| Write energy | 1.2 J/GB | 6 J/GB | 17.5 J/GB [28] |
| Idle power | ~100 mW/GB | ~1 mW/GB | 1-10 mW/GB |
| Density | 1x | 2 - 4x | 4x |

- Compared to NAND Flash, PCM is byte-addressable, has orders of magnitude lower latency and higher endurance.

Sources: [Doller '09] [Lee et al. '09] [Qureshi et al. '09]

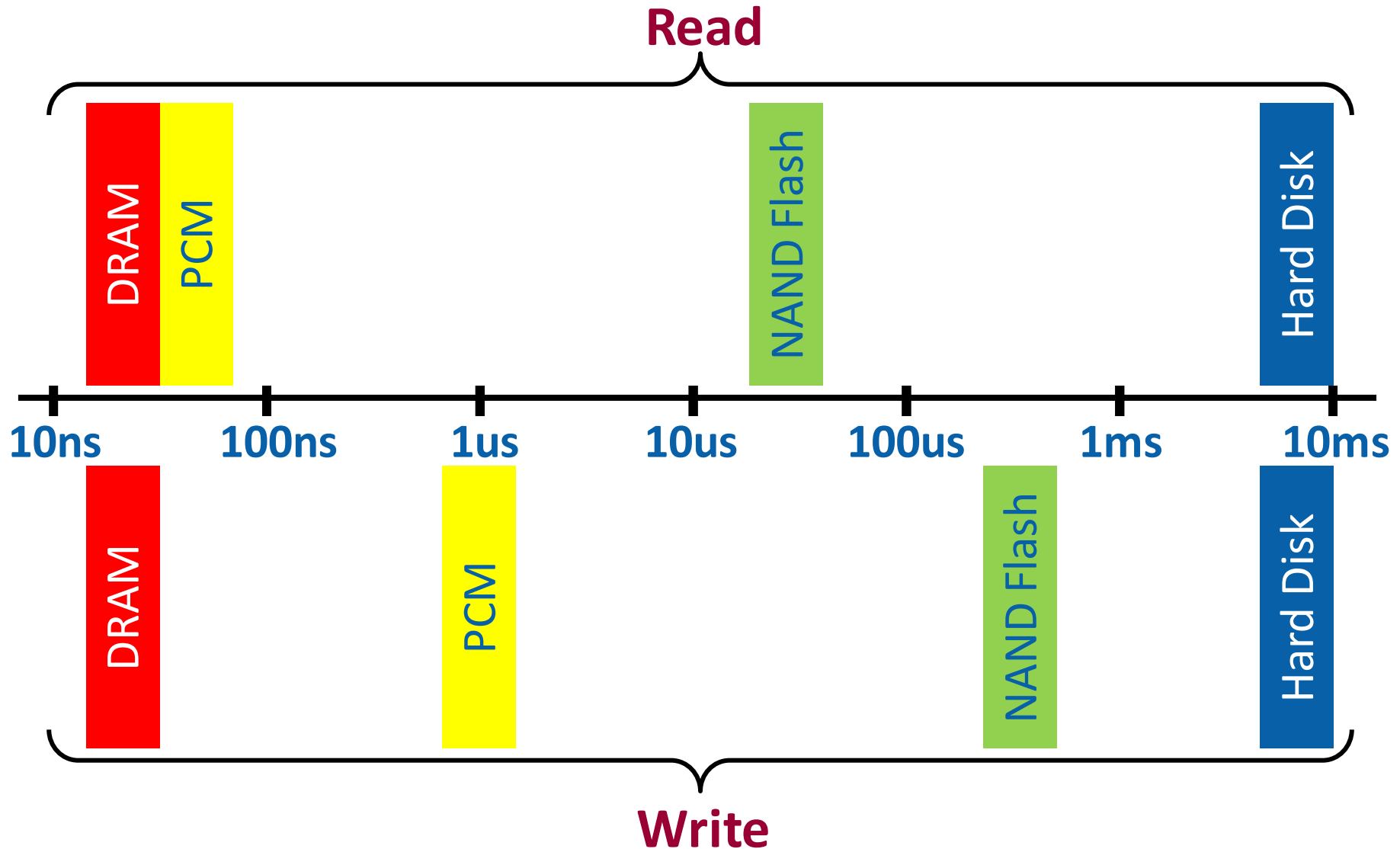
Comparison of Technologies

| | DRAM | PCM | NAND Flash |
|--------------------|------------------|------------------------|----------------------|
| Page size | 64B | 64B | 4KB |
| Page read latency | 20-50ns | ~ 50ns | ~ 25 μ s |
| Page write latency | 20-50ns | ~ 1 μ s | ~ 500 μ s |
| Write bandwidth | ~GB/s per die | 50-100 MB/s per die | 5-40 MB/s per die |
| Erase latency | N/A | N/A | ~ 2 ms |
| Endurance | ∞ | $10^6 - 10^8$ | $10^4 - 10^5$ |
| Read energy | 0.8 J/GB | 1 J/GB | 1.5 J/GB [28] |
| Write energy | 1.2 J/GB | 6 J/GB | 17.5 J/GB [28] |
| Idle power | ~100 mW/GB | ~1 mW/GB | 1-10 mW/GB |
| Density | 1× | 2 – 4× | 4× |

- Compared to DRAM, PCM has better density and scalability; PCM has similar read latency but longer write latency

Sources: [Doller '09] [Lee et al. '09] [Qureshi et al. '09]

Relative Latencies:



Challenge: PCM Writes

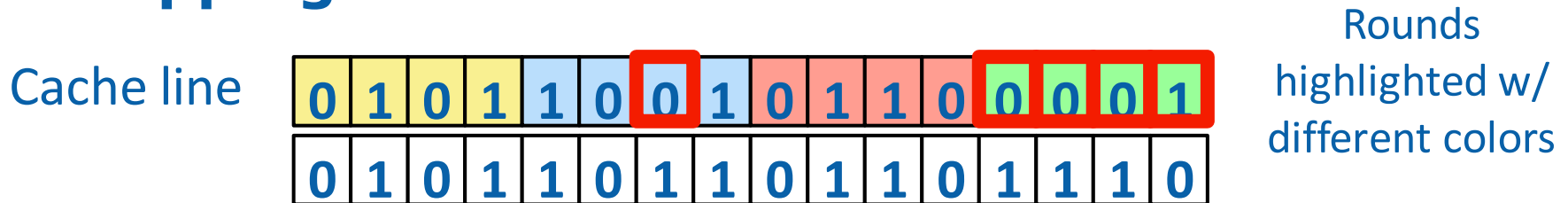
- **Limited endurance**
 - Wear out quickly for hot spots
- **High energy consumption**
 - 6-10X more energy than a read
- **High latency & low bandwidth**
 - SET/RESET time > READ time
 - Limited instantaneous electric current level, requires multiple rounds of writes

| | PCM |
|--------------------|---------------------|
| Page size | 64B |
| Page read latency | ~ 50ns |
| Page write latency | ~ 1 μ s |
| Write bandwidth | 50-100 MB/s per die |
| Erase latency | N/A |
| Endurance | $10^6 - 10^8$ |
| Read energy | 1 J/GB |
| Write energy | 6 J/GB |
| Idle power | ~1 mW/GB |
| Density | 2 – 4× |

PCM Write Hardware Optimization

[Cho, Lee'09] [Lee et al. '09] [Yang et al. '07] [Zhou et al. '09]

- **Baseline: several rounds of writes for a cache line**
 - Which bits in which rounds are hard wired
- **Optimization: data comparison write**
 - Goal: write only modified bits rather than entire cache line
 - Approach: read-compare-write
- **Skipping rounds with no modified bits**



PCM

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

PCM-savvy Algorithms?

New goal: minimize PCM writes

- Writes use 6X more energy than reads
- Writes 20X slower than reads, lower BW, wear-out

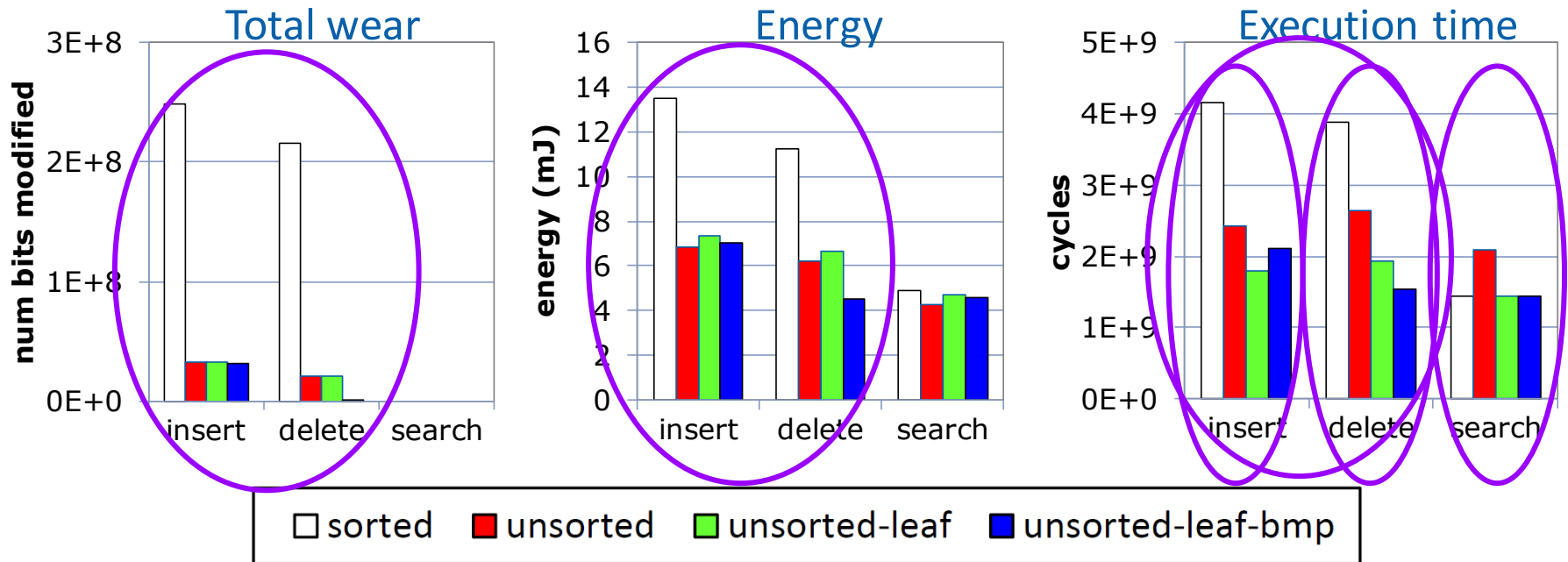
Data comparison writes:

- Minimize Number of bits that change

B⁺-Tree Index

[Chen, G, Nath '11]

Node size 8 cache lines; 50 million entries, 75% full;
Three workloads: Inserting / Deleting / Searching
500K random keys
PTLSSim extended with PCM support



Unsorted leaf schemes achieve the best performance

- For insert intensive: unsorted-leaf
- For insert & delete intensive: unsorted-leaf with bitmap

Multi-core Computing Lectures:

Progress-to-date on Key Open Questions

- **How to formally model multi-core hierarchies?**
- **What is the Algorithm Designer's model?**
- **What runtime task scheduler should be used?**
- **What are the new algorithmic techniques?**
- **How do the algorithms perform in practice?**

References

- [Acar, Blleloch, Blumofe '02] U. A. Acar, G. E. Blleloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Comput. Syst.*, 35(3), 2002
- [Arora et al. '98] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *ACM SPAA*, 1998
- [Bergan et al. '10] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Core-Det: A compiler and runtime system for deterministic multithreaded execution. *ACM ASPLOS*, 2010
- [Berger et al. '09] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. *ACM OOPSLA*, 2009
- [Blleloch, Fineman, G, Shun '12] G. E. Blleloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic algorithms can be fast. *ACM PPOPP*, 2012
- [Blleloch, Reid-Miller '97] G. E. Blleloch and M. Reid-Miller. Pipelining with futures. *ACM SPAA*, 1997
- [Bocchino et al. '09] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. *Usenix HotPar*, 2009
- [Chen, G, Nath '11] S. Chen, P. B. Gibbons, S. Nath. Rethinking database algorithms for phase change memory. *CIDR*, 2011
- [Cheng et al. '98] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. *ACM SPAA*, 1998
- [Cho, Lee'09] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. *IEEE MICRO*, 2009
- [Cole, Ramachandran '12] R. Cole and V. Ramachandran. Efficient resource oblivious algorithms for multicores with false sharing. *IEEE IPDPS* 2012
- [Devietti et al. '11] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDL: A relaxed consistency deterministic computer. *ACM ASPLOS*, 2011
- [Ding et al. '12] Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, Xiaodong Zhang: BWS: balanced work stealing for time-sharing multicores. *EuroSys* 2012

[Doller '09] E. Doller. Phase change memory and its impacts on memory hierarchy. <http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf>, 2009

[Frigo et al. '09] M. Frigo, P. Halpern, C. E. Leiserson, S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. ACM SPAA, 2009

[Halstead '85] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. ACM TOPLAS, 7(4), 1985

[Hower et al. '11] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? Free will to choose. IEEE HPCA, 2011

[Lee et al. '09] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. ACM ISCA, 2009

[Netzer, Miller '92] R. H. B. Netzer and B. P. Miller. What are race conditions? ACM LOPLAS, 1(1), 1992

[Olszewski et al. '09] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. ACM ASPLOS, 2009

[Qureshi et al.'09] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. ACM ISCA, 2009

[Shun, Blelloch, Fineman, G] J. Shun, G. E. Blelloch, J. Fineman, P. B. Gibbons. Priority-write as a parallel primitive. Manuscript, 2012

[Spoonhower, Blelloch, G, Harper '09] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, R. Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. ACM SPAA, 2009

[Steele '90] G. L. Steele Jr. Making asynchronous parallelism safe for the world. ACM POPL, 1990

[Yang et al. '07] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A low power phase-change random access memory using a data-comparison write scheme. IEEE ISCAS, 2007

[Yu and Narayanasamy '09] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. ACM ISCA, 2009

[Zhou et al. '09] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. ACM ISCA, 2009