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
Cilk++ Example: Fibonacci

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

Cilk code

```c
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

```c
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]

```cpp
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

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth first search</td>
<td>Spanning forest</td>
</tr>
<tr>
<td>Suffix array</td>
<td>Minimum spanning forest</td>
</tr>
<tr>
<td>Remove duplicates</td>
<td>Maximal Independent set</td>
</tr>
<tr>
<td>Comparison sort</td>
<td>K-nearest neighbors</td>
</tr>
<tr>
<td>N-body</td>
<td>Triangle ray intersect</td>
</tr>
<tr>
<td>Delaunay triangulation</td>
<td>Delaunay refinement</td>
</tr>
</tbody>
</table>

- **Wide coverage of real-world non-numeric problems**
- **Random numbers can be deterministic**
External vs. Internal Determinism

- **External**: same input $\rightarrow$ same result
- **Internal**: same input $\rightarrow$ 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. \textbf{in parallel do}
3. \{ $r_3 := \text{AtomicAdd}(x, 1)$ \}
4. \{ $r_4 := \text{AtomicAdd}(x, 10)$
5. \textbf{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

<table>
<thead>
<tr>
<th>Functional programming</th>
<th>History-independ. data structures</th>
<th>Deterministic reservations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix array</td>
<td>Remove duplicates</td>
<td>Spanning forest</td>
</tr>
<tr>
<td>Comparison sort</td>
<td>Delaunay refinement</td>
<td>Minimum spanning forest</td>
</tr>
<tr>
<td>N-body</td>
<td></td>
<td>Maximal independent set</td>
</tr>
<tr>
<td>K-nearest neighbors</td>
<td></td>
<td>Breadth first search</td>
</tr>
<tr>
<td>Triangle ray intersect</td>
<td></td>
<td>Delaunay triangulation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delaunay refinement</td>
</tr>
</tbody>
</table>
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**

\[ \text{iterates} = [1,...,n]; \]

\[ \text{while(iterates remain)} \{ \]

- **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;
}

© Phillip B. Gibbons
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

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**

<table>
<thead>
<tr>
<th>Application</th>
<th>Algorithm</th>
<th>1 thread</th>
<th>40 core</th>
<th>$T_1/T_{40}$</th>
<th>$T_S/T_{40}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Sort</td>
<td>serialRadixSort</td>
<td>0.48</td>
<td>0.299</td>
<td>23.0</td>
<td>36.9</td>
</tr>
<tr>
<td></td>
<td>parallelRadixSort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparison Sort</td>
<td>serialSort</td>
<td>2.85</td>
<td>2.59</td>
<td>39.2</td>
<td>43.2</td>
</tr>
<tr>
<td></td>
<td>sampleSort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove Duplicates</td>
<td>serialHash</td>
<td>0.689</td>
<td>0.867</td>
<td>32.1</td>
<td>25.5</td>
</tr>
<tr>
<td></td>
<td>parallelHash</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td>serialHash</td>
<td>0.574</td>
<td>0.748</td>
<td>29.9</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>parallelHash</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth First Search</td>
<td>serialBFS</td>
<td>2.61</td>
<td>5.54</td>
<td>22.4</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td>parallelBFS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spanning Forest</td>
<td>serialSF</td>
<td>1.733</td>
<td>5.12</td>
<td>20.1</td>
<td>6.81</td>
</tr>
<tr>
<td></td>
<td>parallelSF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min Spanning Forest</td>
<td>serialMSF</td>
<td>7.04</td>
<td>14.9</td>
<td>23.8</td>
<td>11.2</td>
</tr>
<tr>
<td></td>
<td>parallelKruskal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Algorithm</th>
<th>1 thread</th>
<th>40 core</th>
<th>$T_1/T_{40}$</th>
<th>$T_S/T_{40}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximal Matching</td>
<td>serialMatching</td>
<td>0.84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>parallelMatching</td>
<td>2.02</td>
<td>0.108</td>
<td>18.7</td>
<td>7.78</td>
</tr>
<tr>
<td>K-Nearest Neighbors</td>
<td>octTreeNeighbors</td>
<td>24.9</td>
<td>1.16</td>
<td>21.5</td>
<td></td>
</tr>
<tr>
<td>Delaunay Triangulation</td>
<td>serialDelaunay</td>
<td>56.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>parallelDelaunay</td>
<td>76.6</td>
<td>2.6</td>
<td>29.5</td>
<td>21.7</td>
</tr>
<tr>
<td>Convex Hull</td>
<td>serialHull</td>
<td>1.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>quickHull</td>
<td>1.655</td>
<td>0.093</td>
<td>17.8</td>
<td>10.9</td>
</tr>
<tr>
<td>Suffix Array</td>
<td>serialKS</td>
<td>17.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>parallelKS</td>
<td>11.7</td>
<td>0.57</td>
<td>20.5</td>
<td>30.4</td>
</tr>
<tr>
<td>Ray Casting</td>
<td>kdtree</td>
<td>7.32</td>
<td>0.334</td>
<td>21.9</td>
<td></td>
</tr>
</tbody>
</table>
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)

\[
\begin{align*}
A &:= 5 & B &:= 17 & B &:= 12 & A &:= 9 & A &:= 8 \\
\text{yields } A = 5 & \text{ and } B = 12
\end{align*}
\]

• **Useful parallel primitive:**
  + Low contention even under high degrees of sharing
  + Avoids many concurrency bugs since commutes
  + Useful for many algorithms & data structures
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 \( E[X_1 + \ldots + X_n] = 1/1+1/2+1/3+\ldots +1/n = H_n \).
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 $= \{\text{source vertex}\}$
In each round:
- In parallel for all $v$ in Frontier
- Remove $v$;
- Attempt to place all $v$’s neighbors in Frontier;

Input: Comb Graph
procedure PRIORITY_WRITE(addr, newval, comp)
    oldval ← *addr
    while comp(newval, oldval) do
        if CAS(addr, oldval, newval) then
            return
        else
            oldval ← *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

![Graph showing running time for remove duplicates](image)
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:**

\[ O(Pd) \text{ steals, } \text{Overheads additive in } \# \text{ of steals} \]

**Beyond Nested Parallelism:**

\[ O(Pd) \text{ steals, } \# \text{ steals can’t bound overheads} \]
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
  – \( \Theta(Pd + Td) \), where \( T \) is # of touches

• Characterize a class of programs using futures effectively
  • Only \( O(Pd) \) deviations
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

\[ \Delta = \# \text{ of deviations} \]

Invocations of slow clones

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

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

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

\[ \forall \text{ computations derived from futures with depth } d \text{ and } T \text{ touches, the expected } \# \text{ deviations by any parsimonious WS scheduler on } P \text{ processors is } O(Pd + Td) \]

- 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: \( \frac{B}{2} \) cache misses incurred by P1 and by P2
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)

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

In-place update


• 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:**

![Diagram showing the phase change memory operations and temperature-current-time relationship.](image)

- **Current** vs. **Temperature**
- **“SET” to Crystalline** at e.g., \(~350^\circ C\)
- **“RESET” to Amorphous** at e.g., \(~610^\circ C\)
- **READ**
Comparison of Technologies

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

- 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

<table>
<thead>
<tr>
<th></th>
<th>DRAM</th>
<th>PCM</th>
<th>NAND Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page size</td>
<td>64B</td>
<td>64B</td>
<td>4KB</td>
</tr>
<tr>
<td>Page read latency</td>
<td>20-50ns</td>
<td>~ 50ns</td>
<td>~ 25 µs</td>
</tr>
<tr>
<td>Page write latency</td>
<td>20-50ns</td>
<td>~ 1 µs</td>
<td>~ 500 µs</td>
</tr>
<tr>
<td>Write bandwidth</td>
<td>~GB/s per die</td>
<td>50-100 MB/s per die</td>
<td>5-40 MB/s per die</td>
</tr>
<tr>
<td>Erase latency</td>
<td>N/A</td>
<td>N/A</td>
<td>~ 2 ms</td>
</tr>
<tr>
<td>Endurance</td>
<td>∞</td>
<td>10⁶ – 10⁸</td>
<td>10⁴ – 10⁵</td>
</tr>
<tr>
<td>Read energy</td>
<td>0.8 J/GB</td>
<td>1 J/GB</td>
<td>1.5 J/GB [28]</td>
</tr>
<tr>
<td>Write energy</td>
<td>1.2 J/GB</td>
<td>6 J/GB</td>
<td>17.5 J/GB [28]</td>
</tr>
<tr>
<td>Idle power</td>
<td>~100 mW/GB</td>
<td>~1 mW/GB</td>
<td>1–10 mW/GB</td>
</tr>
<tr>
<td>Density</td>
<td>1×</td>
<td>2 – 4×</td>
<td>4×</td>
</tr>
</tbody>
</table>

- 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:

Read

Write

10ns  100ns  1us  10us  100us  1ms  10ms

DRAM  PCM  NAND Flash  Hard Disk

DRAM  PCM  NAND Flash  Hard Disk
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

---

<table>
<thead>
<tr>
<th></th>
<th>PCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page size</td>
<td>64B</td>
</tr>
<tr>
<td>Page read latency</td>
<td>~50ns</td>
</tr>
<tr>
<td>Page write latency</td>
<td>~1 μs</td>
</tr>
<tr>
<td>Write bandwidth</td>
<td>50-100 MB/s per die</td>
</tr>
<tr>
<td>Erase latency</td>
<td>N/A</td>
</tr>
<tr>
<td>Endurance</td>
<td>$10^6 - 10^8$</td>
</tr>
<tr>
<td>Read energy</td>
<td>1 J/GB</td>
</tr>
<tr>
<td>Write energy</td>
<td>6 J/GB</td>
</tr>
<tr>
<td>Idle power</td>
<td>~1 mW/GB</td>
</tr>
<tr>
<td>Density</td>
<td>2 – 4×</td>
</tr>
</tbody>
</table>
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**

```plaintext
Cache line: 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1
PCM: 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1
```

Rounds highlighted w/ different colors
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


