Multi-core Computing
Lecture 1

MADALGO Summer School 2012
Algorithms for Modern Parallel and Distributed Models

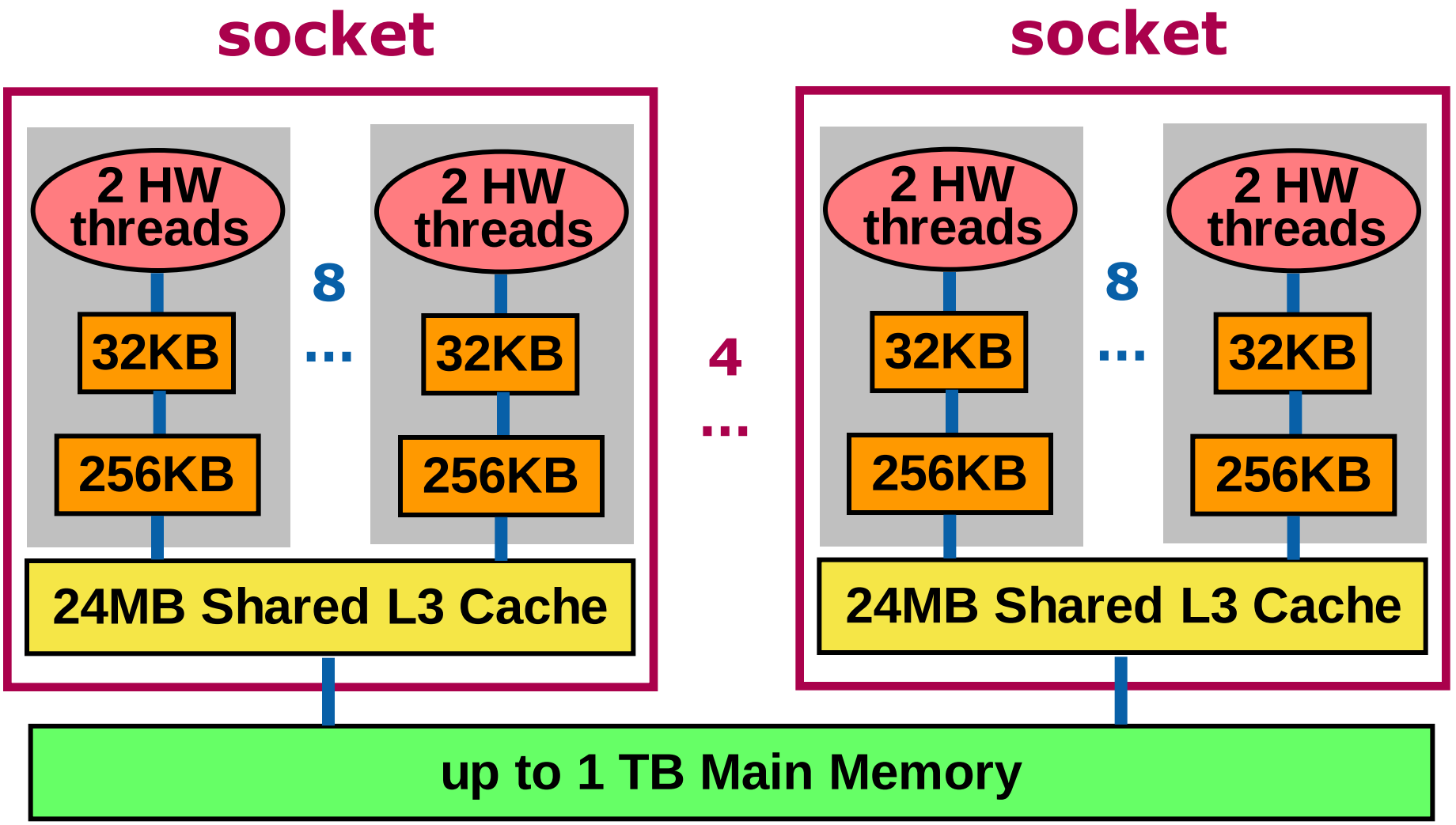
Phillip B. Gibbons
Intel Labs Pittsburgh

August 20, 2012

Lecture 1 Outline

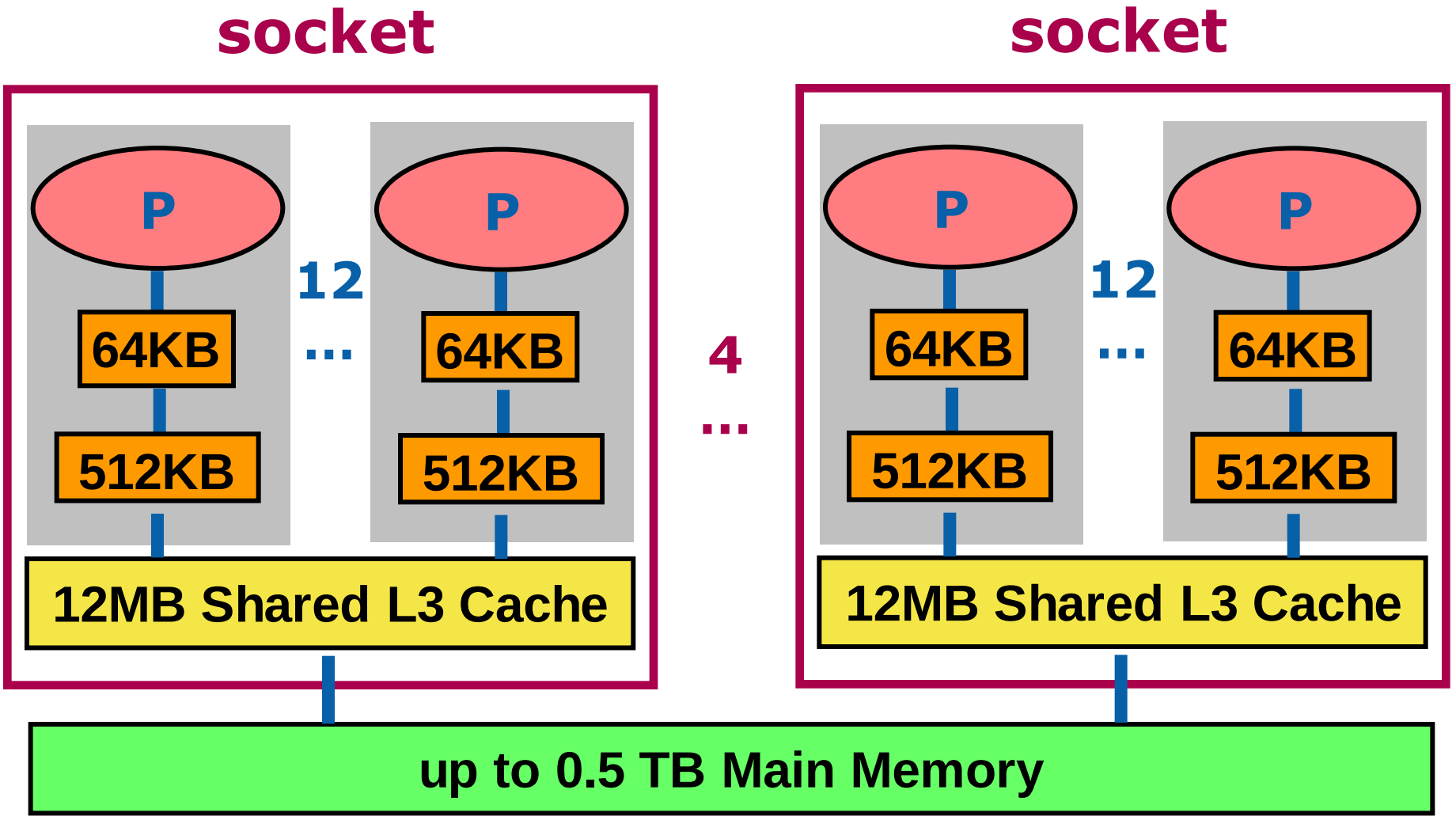
- **Multi-cores: today, future trends, challenges**
- **Computations & Schedulers**
  - Modeling computations in work-depth framework
  - Schedulers: Work Stealing & PDF
- **Cache miss analysis on 2-level parallel hierarchy**
  - Private caches OR Shared cache
- **Low-depth, cache-oblivious parallel algorithms**
  - Sorting & Graph algorithms

32-core Xeon 7500 Multi-core

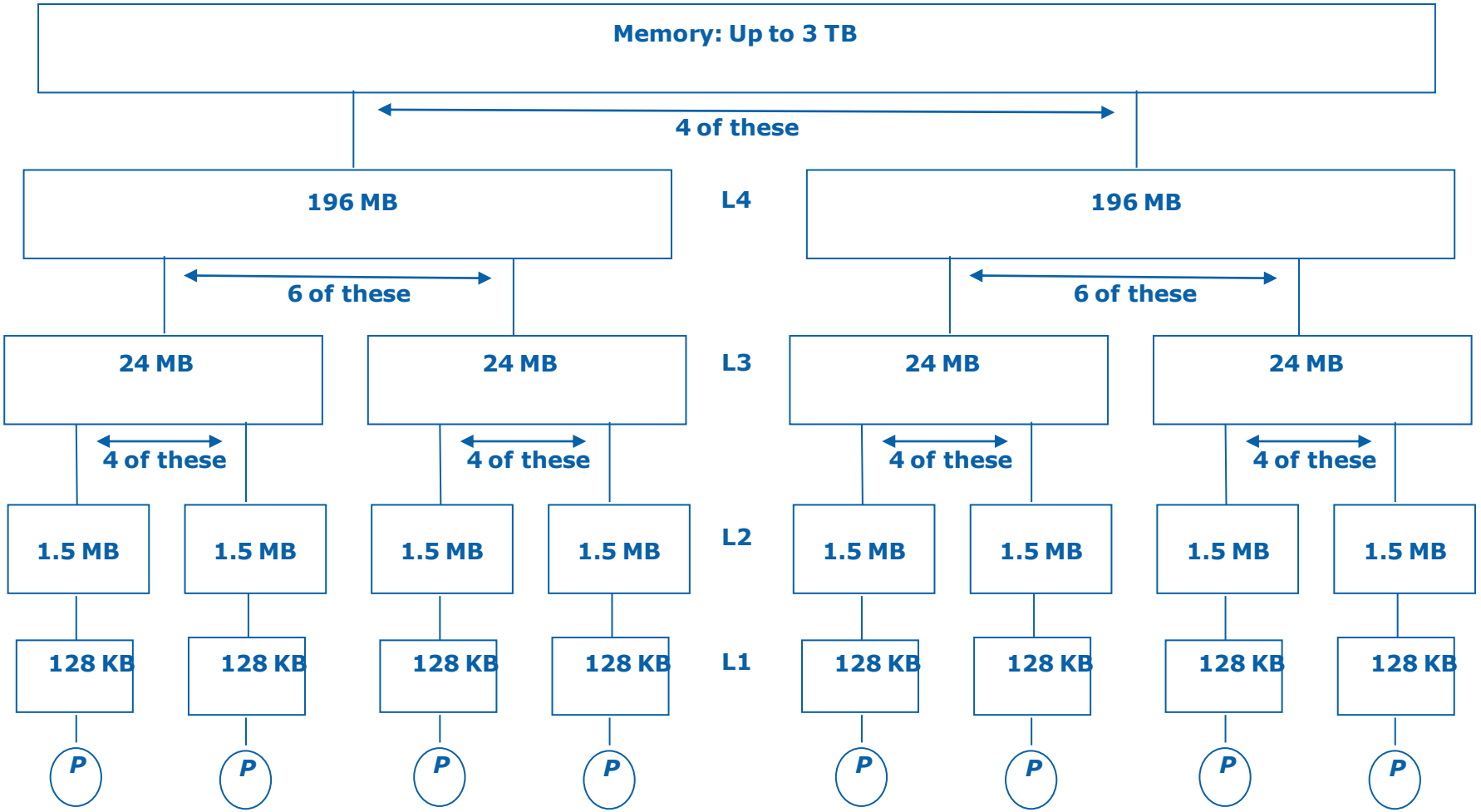


Attach: Magnetic Disks & Flash Devices

48-core AMD Opteron 6100



96-core IBM z196 Multi-core

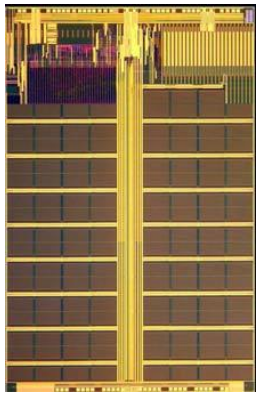


4 levels of cache

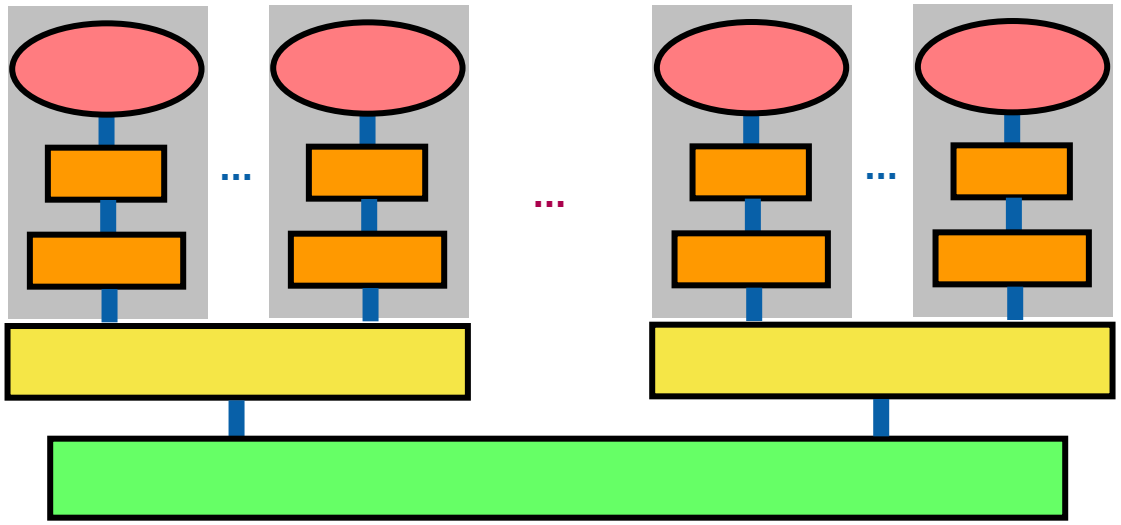
Hierarchy Trends

- **Good performance [energy] increasingly requires effective use of hierarchy**

- **Hierarchy getting richer**
  - More cores
  - More levels of cache
  - New memory/storage technologies
    - Flash/SSDs, emerging PCM
    - Bridge gaps in hierarchies – can't just look at last level of hierarchy



Harder to Use Hierarchy Effectively



Challenges

- Cores compete for hierarchy
- Hard to reason about parallel performance
- Hundred cores coming soon
- Cache hierarchy design in flux
- Hierarchies differ across platforms

How Hierarchy is Treated Today

Algorithm Designers & Application/System Developers tend towards one of two extremes

API view:
Memory + I/O;
Parallelism often ignored

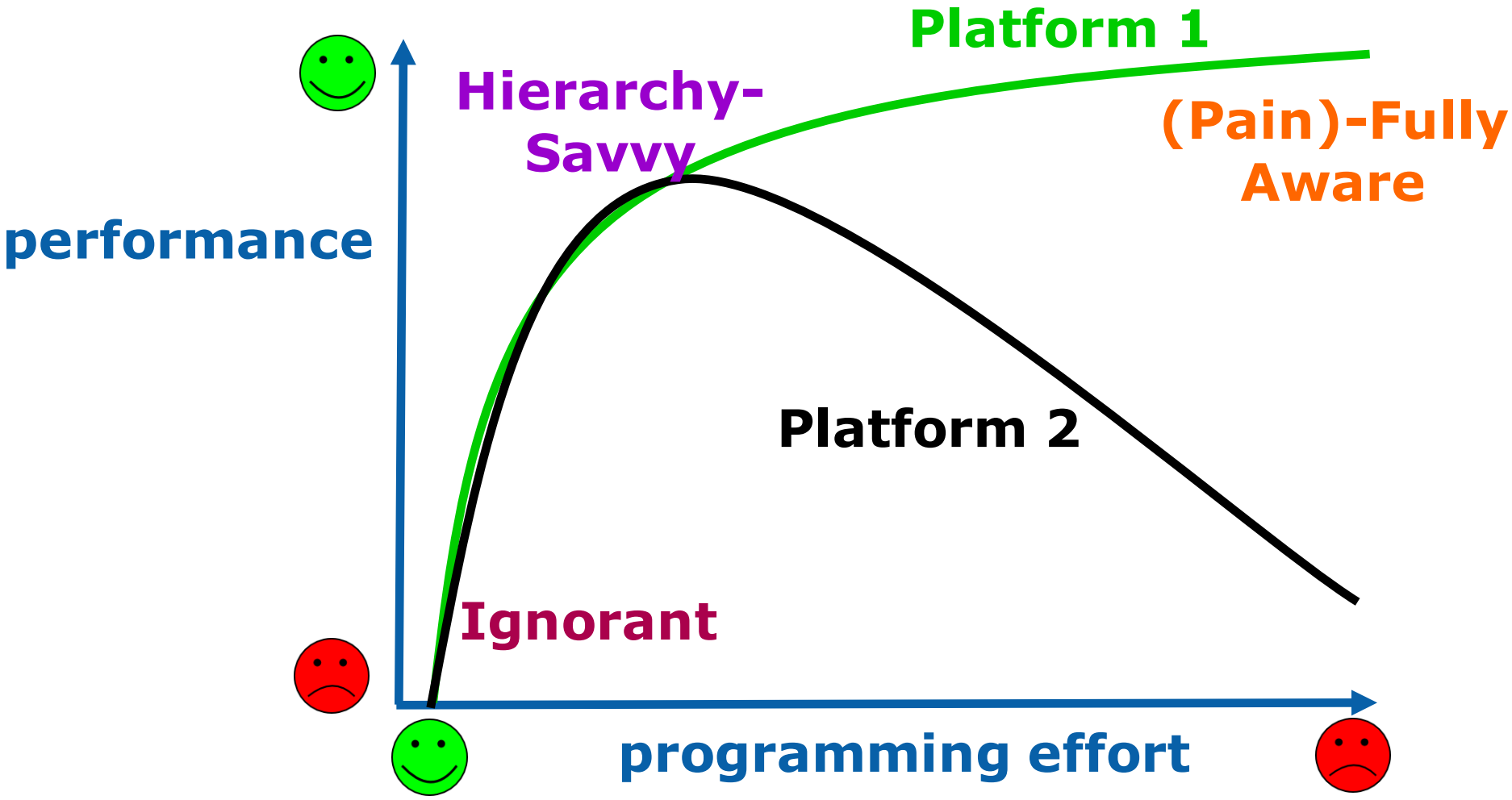
➡ **Performance iffy**

Hand-tuned to platform

➡ **Effort high,**
Not portable,
Limited sharing scenarios

Or they focus on one or a few aspects, but without a comprehensive view of the whole

Hierarchy-Savvy Sweet Spot



Goal: Modest effort, good performance, robust, strong theoretical foundation

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?**

Focus: Irregular Algorithms

- Sequences and strings: Sorting, Suffix arrays, Seq. alignment
- Graph algorithms: Min spanning tree, BFS, coloring, separators
- Machine learning: Sparse SVM, K-means, Gibbs sampling, LASSO
- Graphics: Ray tracing, Micropoly rendering
- Geometry: Delaunay triangulation, Nearest neighbors, N-body

Compared to well-studied regular algorithms:

- Harder to find effective parallelism
- Harder to exploit memory hierarchy

Lecture 1 Outline

- **Multi-cores: today, future trends, challenges**

- **Computations & Schedulers**
  - Modeling computations in work-depth framework
  - Schedulers: Work Stealing & PDF

- **Cache miss analysis on 2-level parallel hierarchy**
  - Private caches OR Shared cache

- **Low-depth, cache-oblivious parallel algorithms**
  - Sorting & Graph algorithms

Coarse- vs. Fine-grain threading

- **Coarse Threading popular for decades**
  - Spawn one thread per core at program initialization
  - Heavy-weight O.S. threads
  - E.g., Splash Benchmark

- **Better Alternative:**
  - System supports user-level light-weight threads
  - Programs expose lots of parallelism
  - Dynamic parallelism: forking can be data-dependent
  - Smart runtime scheduler maps threads to cores, dynamically as computation proceeds

e.g., Cilk++, Intel TBB, OpenMP, MS Parallel Task Library

PRAM vs. Work-Depth Framework

PRAM [Fortune, Wyllie '78]

- P processors, Shared memory, Synchronous
- Algorithm specifies what each processor does at each synchronous step

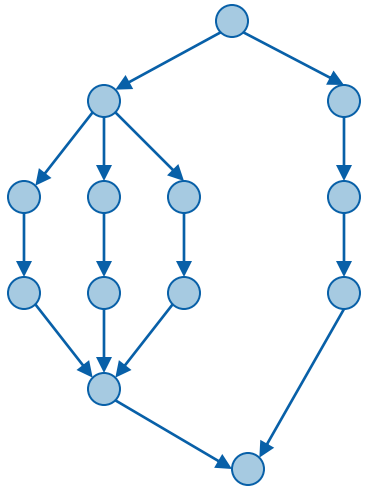
Work-Depth framework [Shiloach, Vishkin '82]

- Shared memory, Synchronous
- Algorithm specifies what tasks (work) can be done in parallel at each synchronous step
- Work W is number of tasks
- Depth D is number of steps
- Brent's "Theorem" ['74]: On P processors,
  Time ≤ W/P + D

Work-Depth Generalized to Nested Parallel Computations

Computation DAG:

- Nodes are tasks
- Edges are dependences between tasks

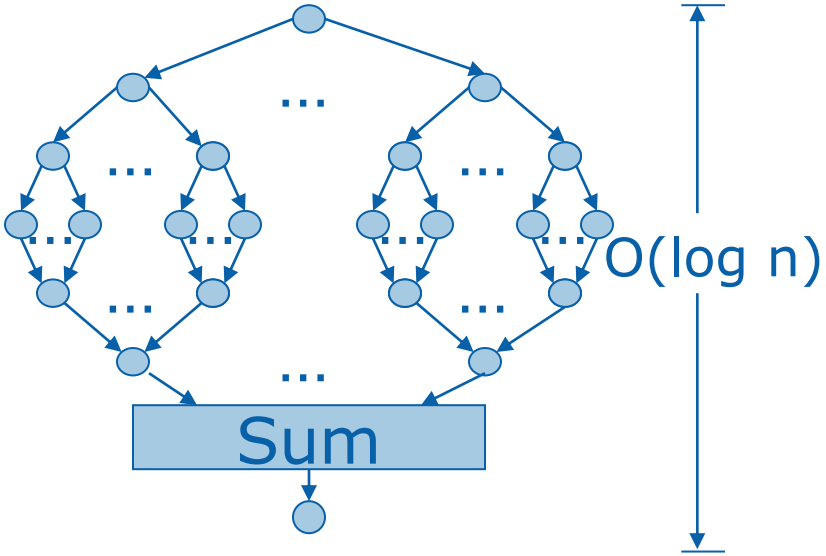


Common Assumptions:

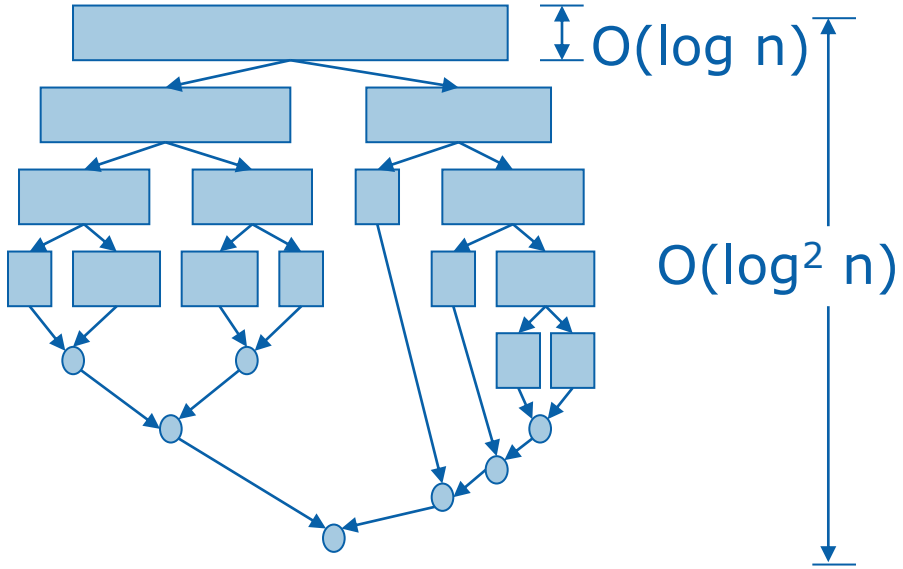
1. Structure may depend on input: revealed online
2. Nested-parallel (fork/join): DAG is series-parallel
3. Structure is independent of the parallel schedule

Work = sequential execution time
Depth = span = critical path length

Example Computation DAGs



Matrix Multiply

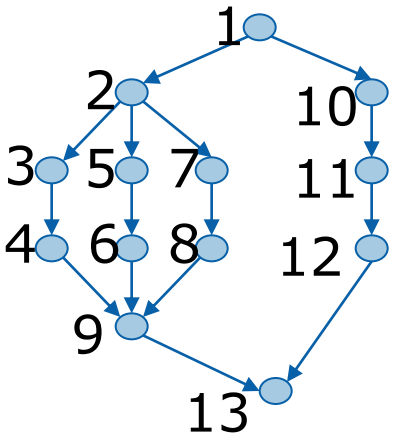


Quicksort variant

Sequential Schedule

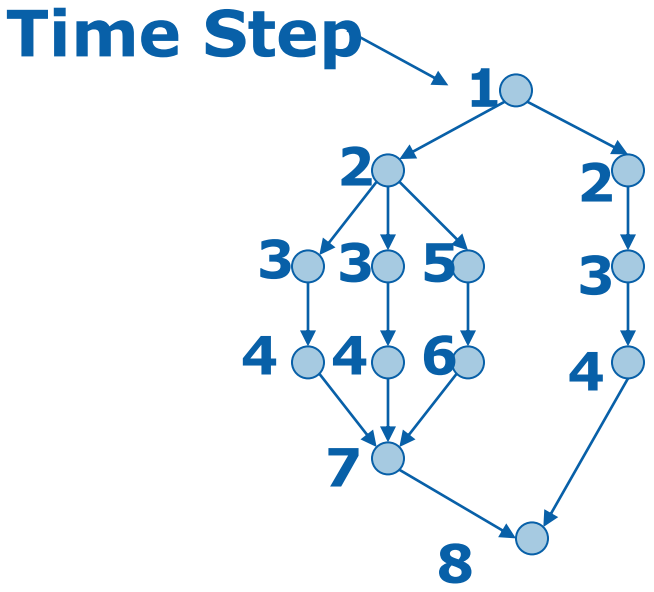
- **Any topological sort of the DAG**

Example: sequential depth-first schedule



Parallel Schedules

Up to P tasks executed on each time step



a 3 processor schedule

Greedy Schedule: if there are ready tasks, do them.

Will guarantee:

- **Time ≤ W/P + D**
- **within a factor of 2 of optimal**

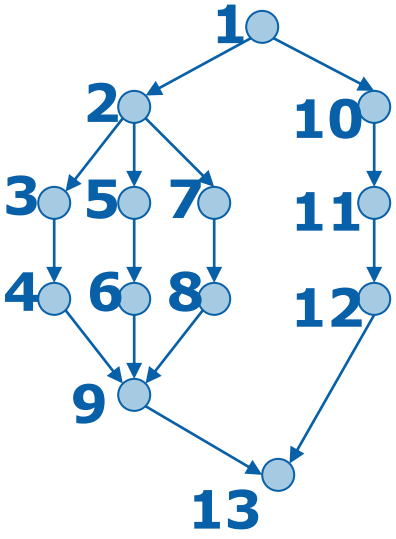
Work Stealing Parallel Scheduler

- **WS = work stealing as in Cilk, elsewhere**
  - **Forked tasks placed on local work queue**
  - **Processor works from bottom of local queue**
  - **When local queue empty, steal top task from random queue**

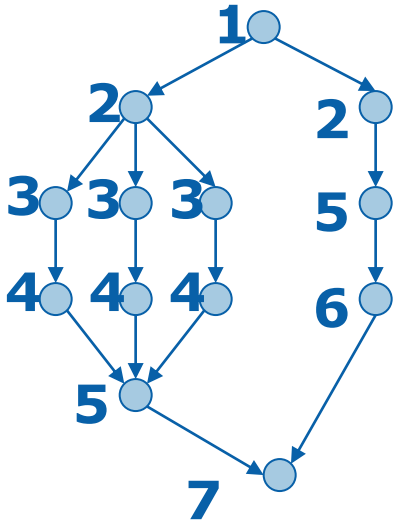
- **Bound number of steals using WS**
  - **Thrm: Expected O(P D) steals for nested-parallel DAG of depth D, on P processors** [Blumofe, Leiserson '99]
    - Proof intuitively shows that every O(P) steals, depth remaining must decrease by ≥1

Prioritized Schedules & PDF

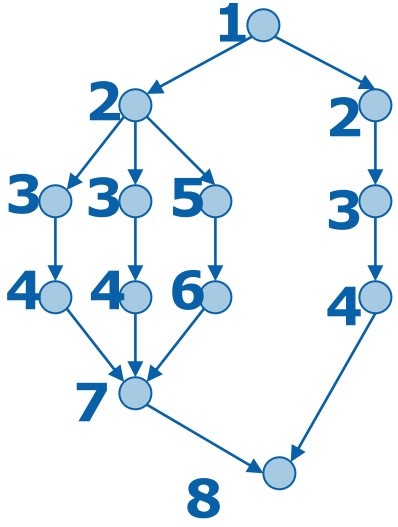
- **Among ready nodes, always give priority to earliest in the sequential schedule**



Sequential



Prioritized



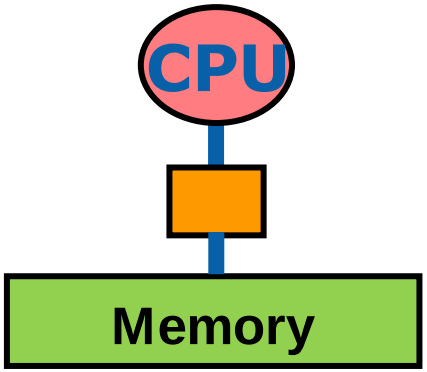
Not prioritized (from before)

Parallel Depth-First (PDF) [Blelloch, G, Matias '99]: prioritized based on depth-first sequential schedule

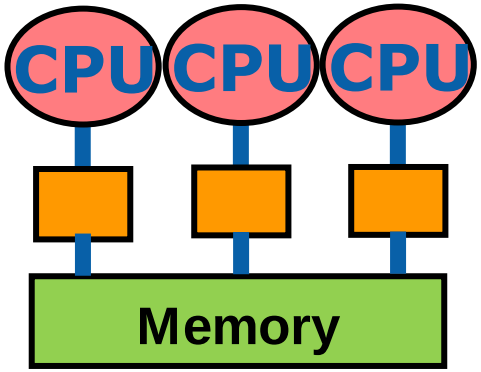
Lecture 1 Outline

- **Multi-cores: today, future trends, challenges**
- **Computations & Schedulers**
  - Modeling computations in work-depth framework
  - Schedulers: Work Stealing & PDF
- **Cache miss analysis on 2-level parallel hierarchy**
  - Private caches OR Shared cache
- **Low-depth, cache-oblivious parallel algorithms**
  - Sorting & Graph algorithms

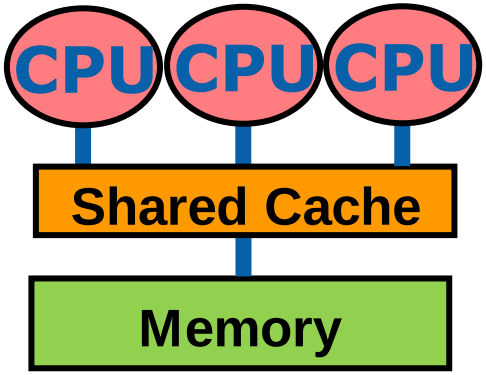
Cache Misses: Simplified Scenarios



Sequential: cache of size $\mathbf{M_1}$



Private caches of size $\mathbf{M_1}$ each



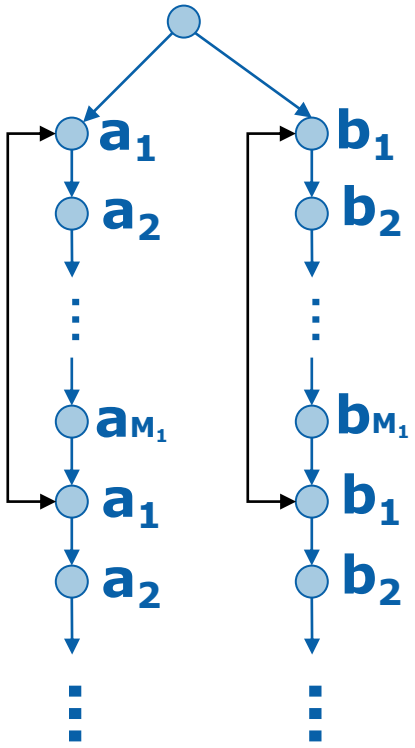
Shared cache of size $\mathbf{M_p}$

- **One level of caches, block size = 1**
- **Ideal replacement policy**

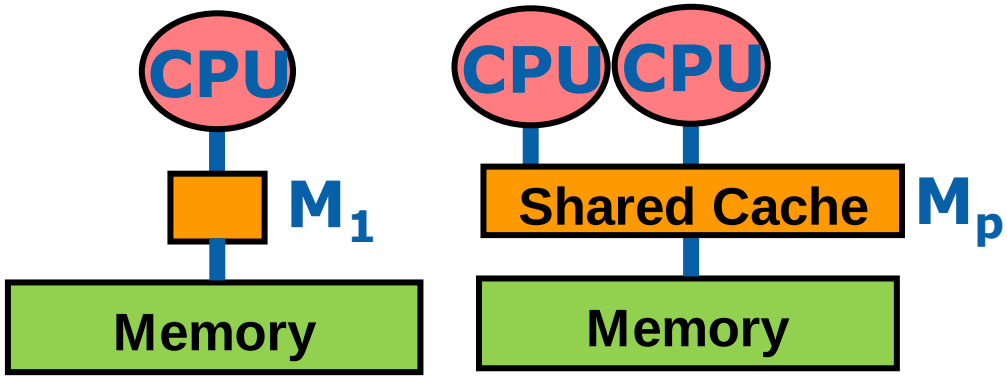
Number of cache misses depends on the schedule

Parallel Schedule with Many Misses

Reads same location M_1 apart



Repeat n times



Sequential schedule has
- 2 M_1 misses

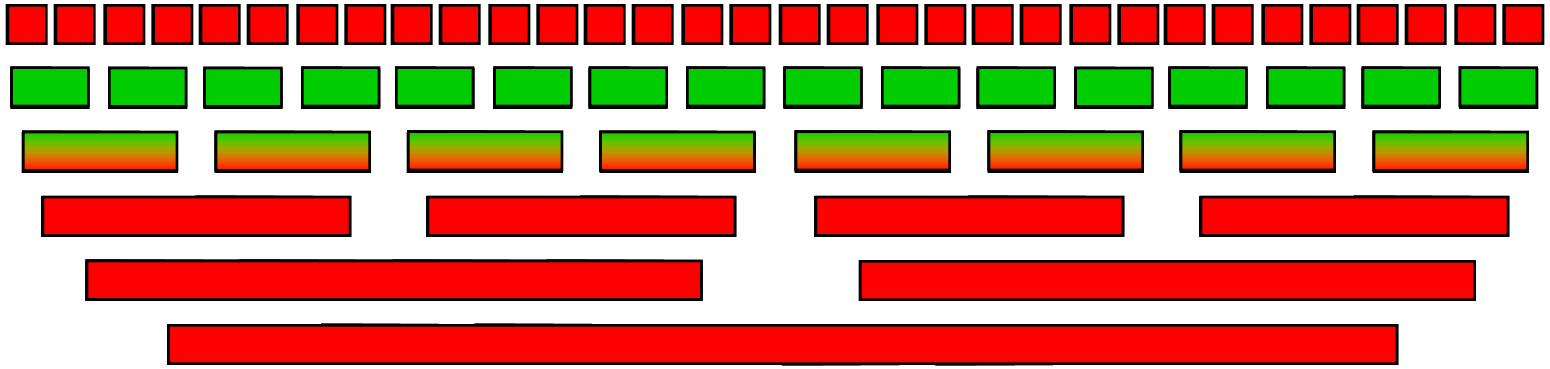
Parallel schedule (P=2):

…when $M_p = M_1$ has
- n M_1 misses

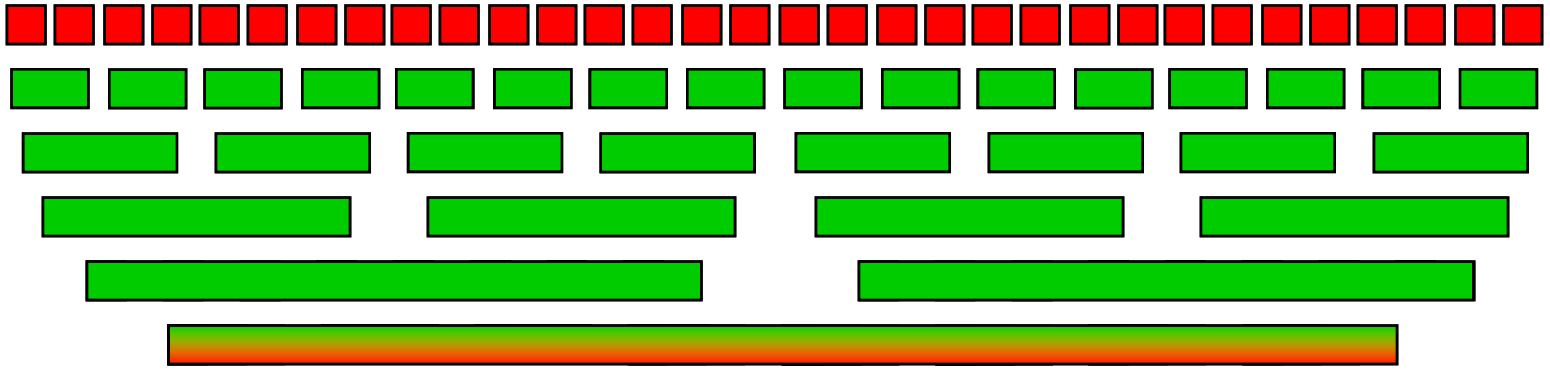
…when $M_p = 2M_1$ has
- 2 M_1 misses only

Parallel Merge Sort: WS vs. PDF

Work Stealing (WS): 8 cores

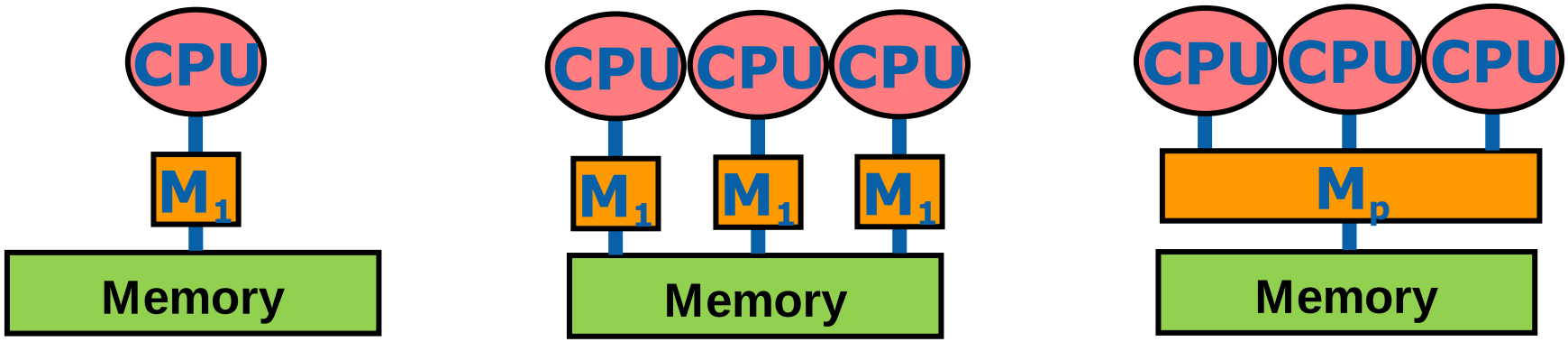


Parallel Depth First (PDF):



Shared cache = 0.5 *(src array size + dest array size).

WS Cache Bounds



Private caches: [Blumofe et al. '96; Acar, Blelloch, Blumofe '02]

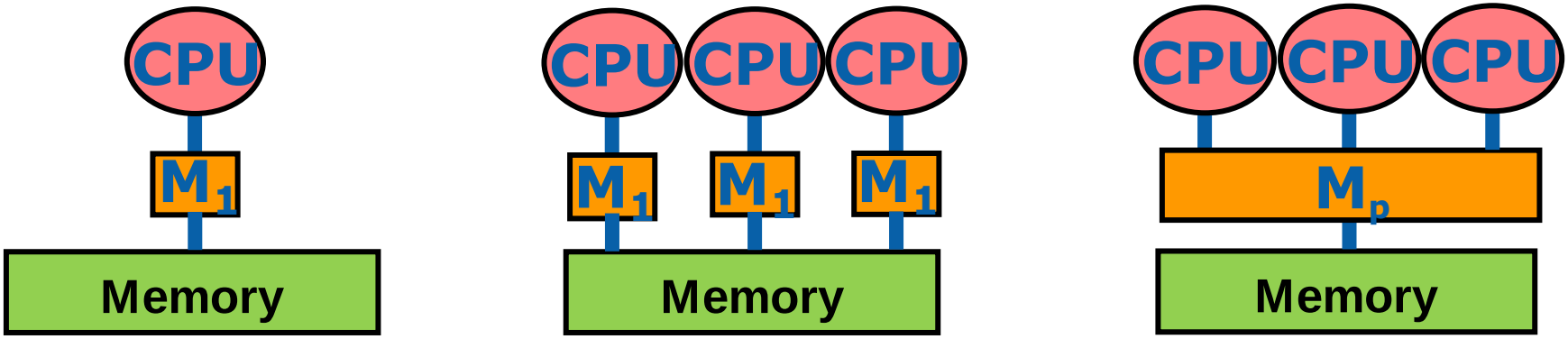
- **Only $O(P\ D\ M_1)$ more misses than sequential**
  - **Proof idea: Each of O(PD) steals incurs $O(M_1)$ extra misses**
  - **Assumes DAG consistency among caches**

Shared Cache:

- **$M_p = P\ M_1$ for no more misses than sequential**

(Bounds are tight. D is the worst case running time of critical path)

PDF Cache Bounds



Private caches:

- **Exercise: How bad can it be?**

Shared Cache: [Blelloch, G '04]

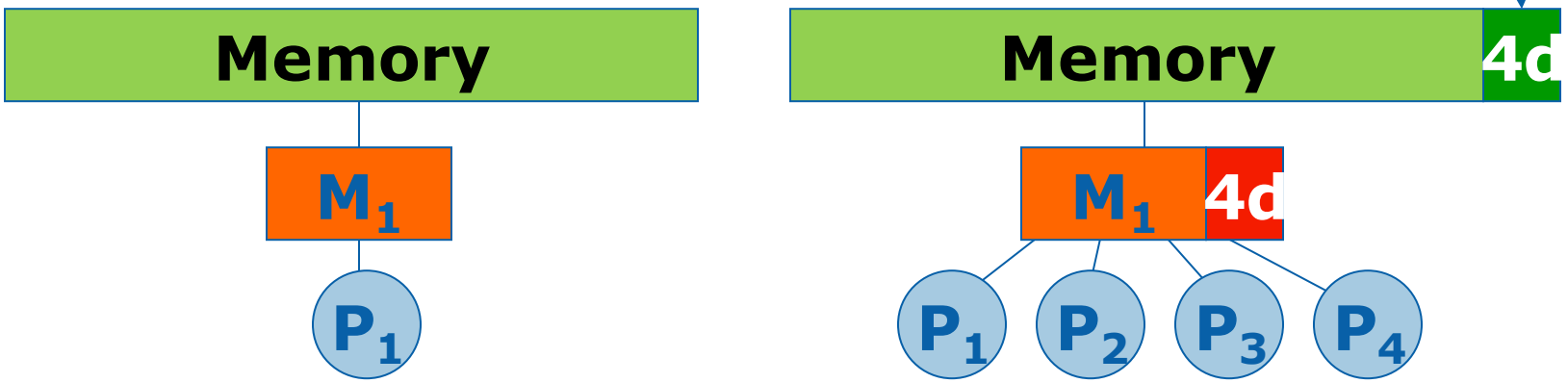
- **Only $M_p = M_1 + P\ D$ (vs. $M_p = P\ M_1$ for WS) for no more misses than sequential**

(Bound is tight. D is the worst case running time of critical path)

PDF bound Paraphrased

- For a program with **sufficient available parallelism**, we only need a **slightly larger shared cache** for P processors than for 1 processor.

[Blelloch, G, Matias '99]

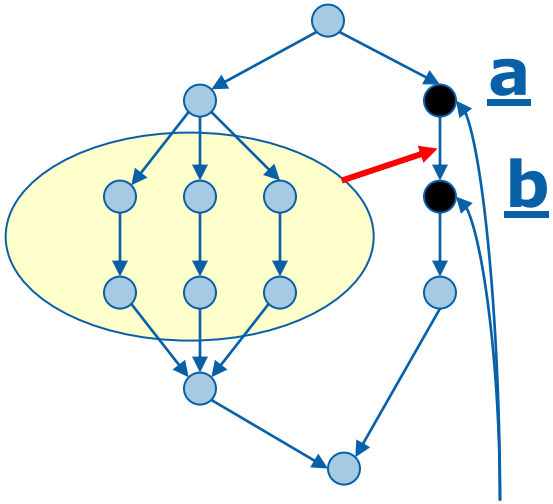


E.g., Matrix Multiply: $M_P = M_1 + O(P \log n)$
Quicksort: $M_p = M_1 + O(P \log^2 n)$

Proof Sketch of $M_p = M_1 + P\ D$ result

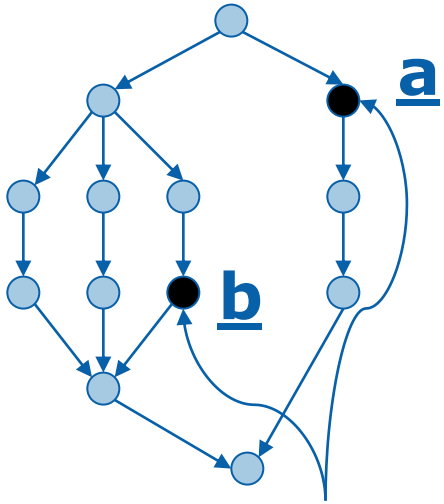
Causes of additional cache misses

Separated execution



Read the same location

Out of order execution



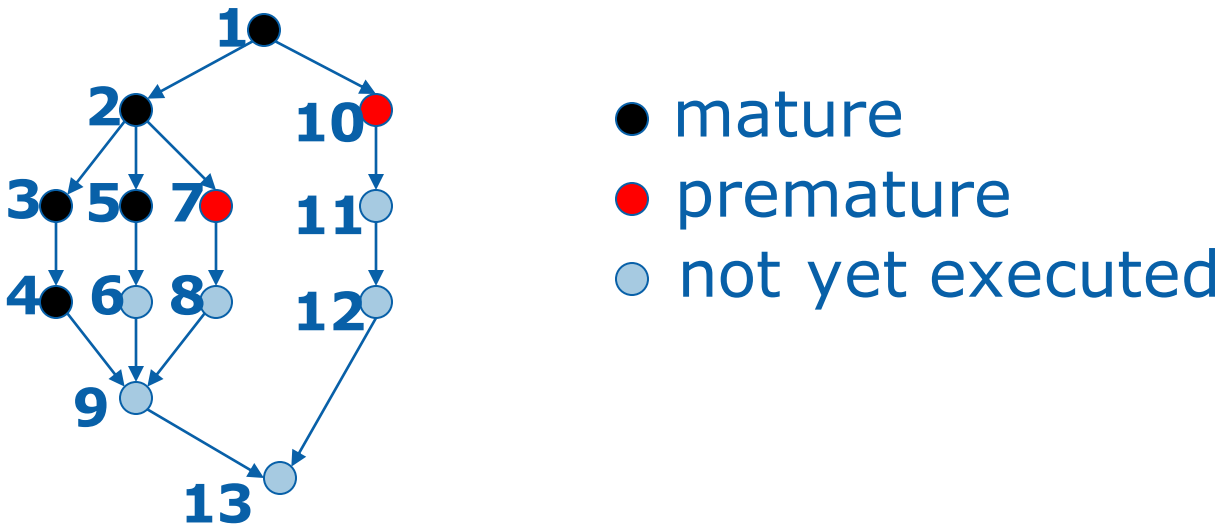
Read the same location

Both are cases of a executing "prematurely"

Premature Nodes

A node is premature at a point in the schedule if:

1. **if it has been executed, and**
2. **there is a node with higher priority that has not yet been executed.**



Premature Node Lemma

PREMATURE NODE LEMMA [Blelloch, G, Matias '99]:

At any point during a prioritized schedule at most PD nodes are premature

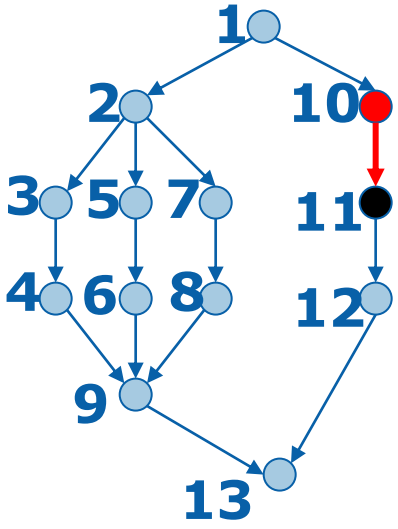
Sufficient to bound extra memory to kPD if every node allocates at most k memory.

But note that:

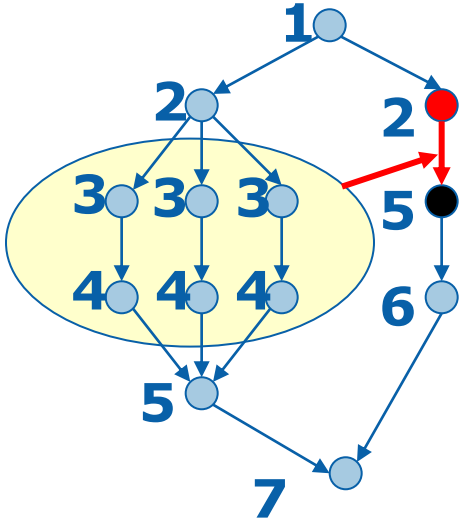
1. **Every premature node can cause an additional cache miss**
2. **Total number of premature nodes over time is not bounded by PD; it could be much larger**

Proof Sketch of $M_p = M_1 + P\,D$ result: Basic Argument (case 1)

Show that value will still be in cache for the delayed nodes (hold premature node's value in extra cache space)



Sequential Order

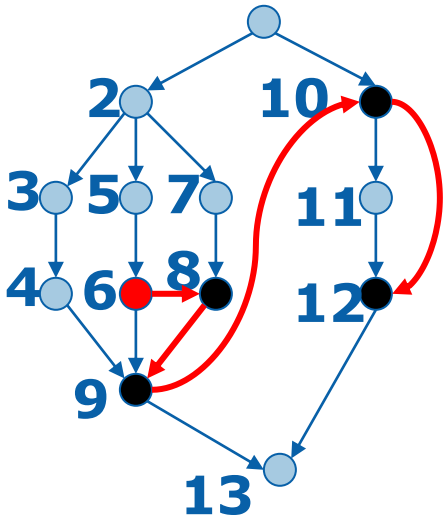


Prioritized Parallel Order

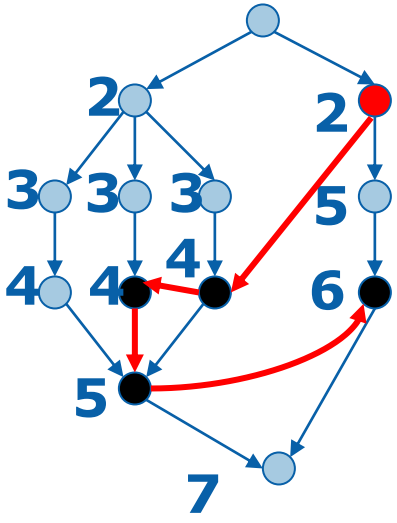
Basic Argument (case 2)

…But still need to account for misses caused by executing a premature node itself

Show that for every additional miss caused by a premature node there is previous miss that becomes a hit



Sequential Order

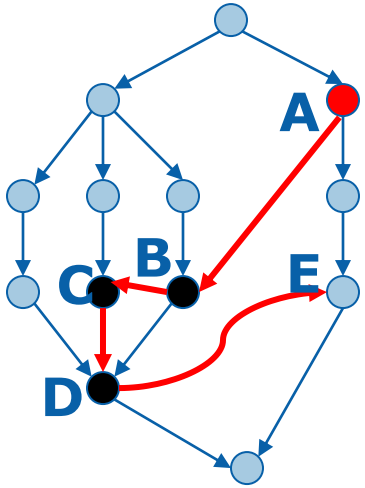


Prioritized Parallel Order

Case 2 sketch (by example)

Imagine reserving PD of cache just for premature nodes—moved to regular cache when they become mature.

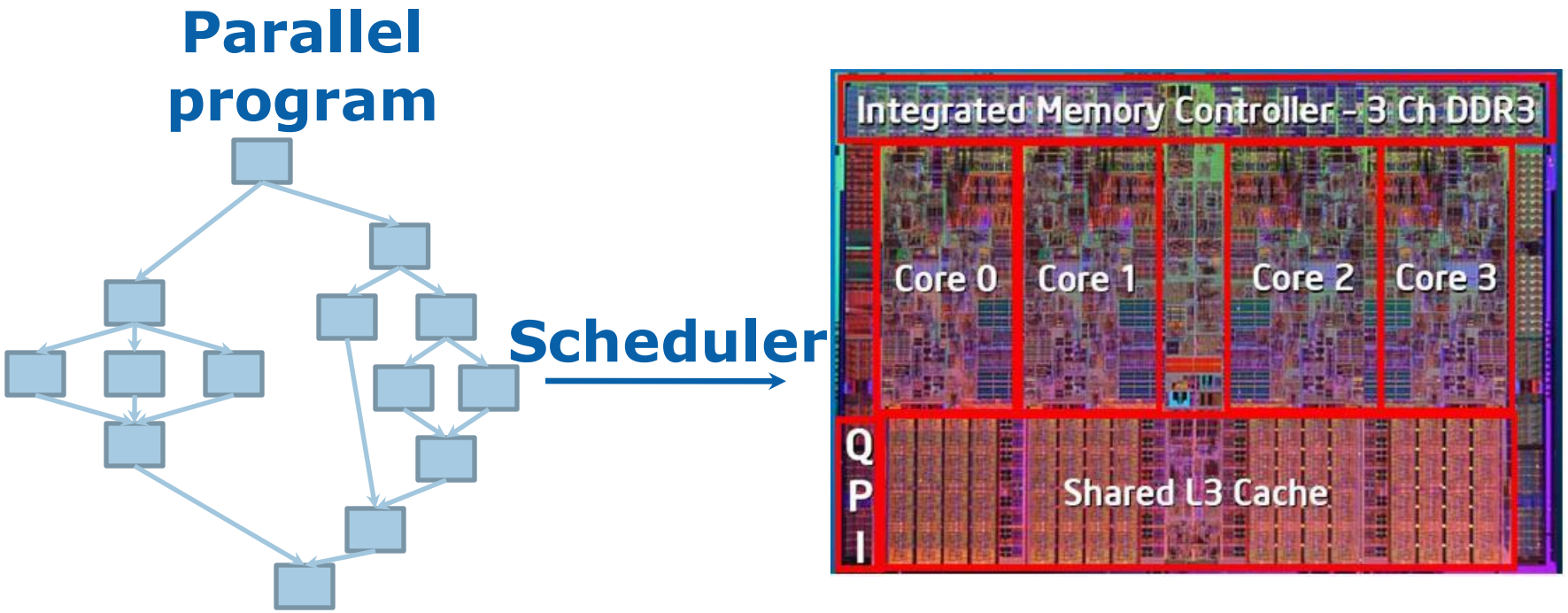
1. As long as A has executed, but not node D, A is in the "special cache"
2. When A becomes mature, prioritize according to sequential order with respect to mature nodes
3. Will be there for E since we have M_1 "regular cache" and any premature nodes go to the special cache



Lecture 1 Outline

- **Multi-cores: today, future trends, challenges**
- **Computations & Schedulers**
  - Modeling computations in work-depth framework
  - Schedulers: Work Stealing & PDF
- **Cache miss analysis on 2-level parallel hierarchy**
  - Private caches OR Shared cache
- **Low-depth, cache-oblivious parallel algorithms**
  - Sorting & Graph algorithms

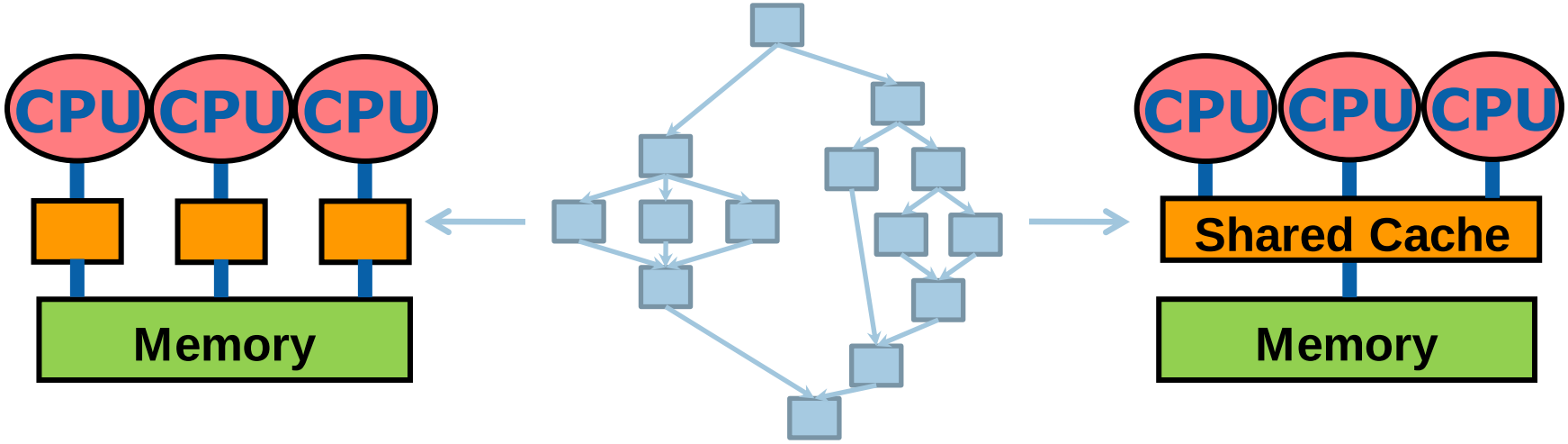
Back to the Big Picture



- Programmer writes high level parallel code
- Ideally, this is portable to different machines
- A run time scheduler dynamically schedules program tasks on to the processors

Objective

- **Design portable parallel programs that can be scheduled to run fast on different kinds of machines**



- Programmer should be able to obtain precise bounds based on certain metrics (like work, depth, cache/memory cost) of the program, and the choice of scheduler

Low-depth -> Good Cache Bounds

- **WS on Private caches:**
  - Only $O(P\ D\ M_1)$ more misses than sequential
- **PDF on Shared cache:**
  - $M_p = M_1 + P\ D$ for no more misses than sequential

Important reason to have algorithm parallelism far exceed number of processors
W/P + D (but still want D << W/P)

Approach

Design a CO algorithm that:

is cache efficient under sequential execution

has low depth (polylogarithmic)

For a parallel system, use an appropriate scheduler that converts:

Good sequential cache complexity

To

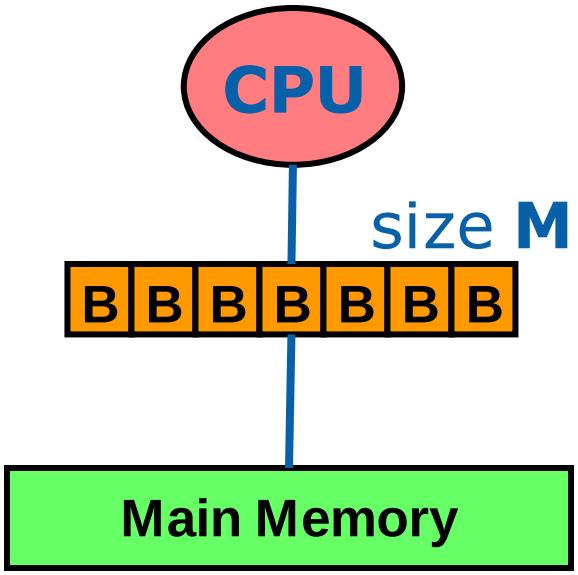
Good parallel cache complexity

Good parallel cache complexity

Sequential Cache Complexity

Ideal Cache Model [Frigo, Leiserson, Prokop, Ramachandran '99]

- **Analysis: Bound the number of cache misses $Q_A(n;M,B)$ of a sequential execution of**
  - Algorithm **A** on input of size **n**
  - "ideal" cache of size **M** organized into
  - blocks of size **B**



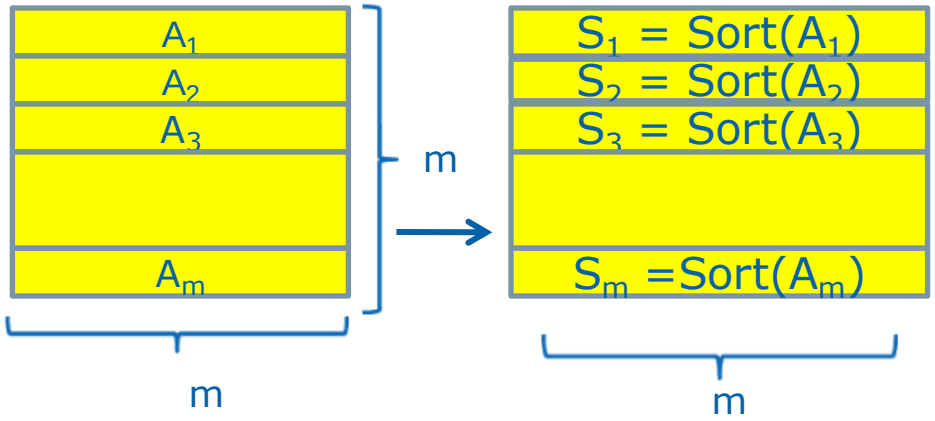
Cache Oblivious Algorithms

- **Algorithm A is designed w/o prior knowledge of cache or block size**
- **Function Q_A is "smooth"**
- **Number of cache misses is roughly Q_A on a realistic cache, same function for any number of cache levels**

BT-Sort: [Blelloch, G, Simhadri '10]

A poly-log depth, cache-oblivious sample sort

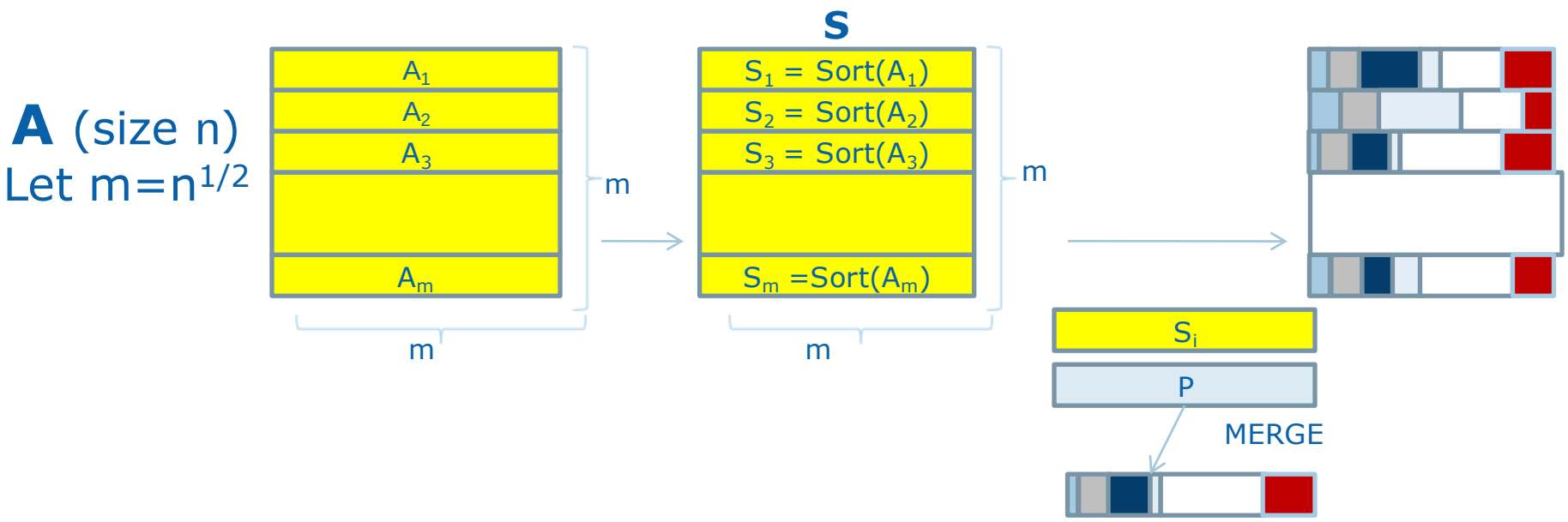
A (size n)
Let $m=n^{1/2}$



- **A is a packed array of size n**

1. **Divide A into m rows & recursively sort each row to get S**
2. **Pick every $\log(n)^{th}$ element from S (call this set X. |X| = n/log n)**
3. **Y = mergesort(X)** **Note: O(n/B) cache complexity**

BT-Sort

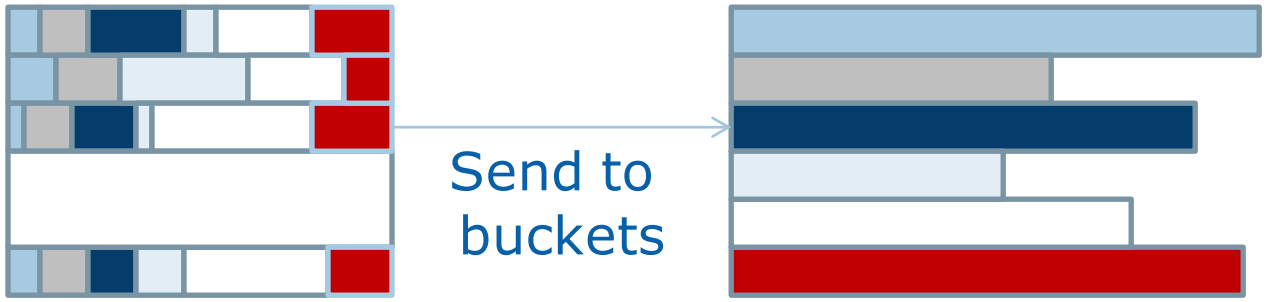


4. Pick every $(n^{1/2}/\log(n))^{th}$ element from Y, call this P (pivots). $|P| = n^{1/2}$

5. Use CO merging algorithm (on (P, S_i)) to partition each of S_i into segments S_{ij} (elements of S_i that go to bucket j)

6. Can compute offset of each segment in bucket using CO prefix sum

7. Bucket Transpose

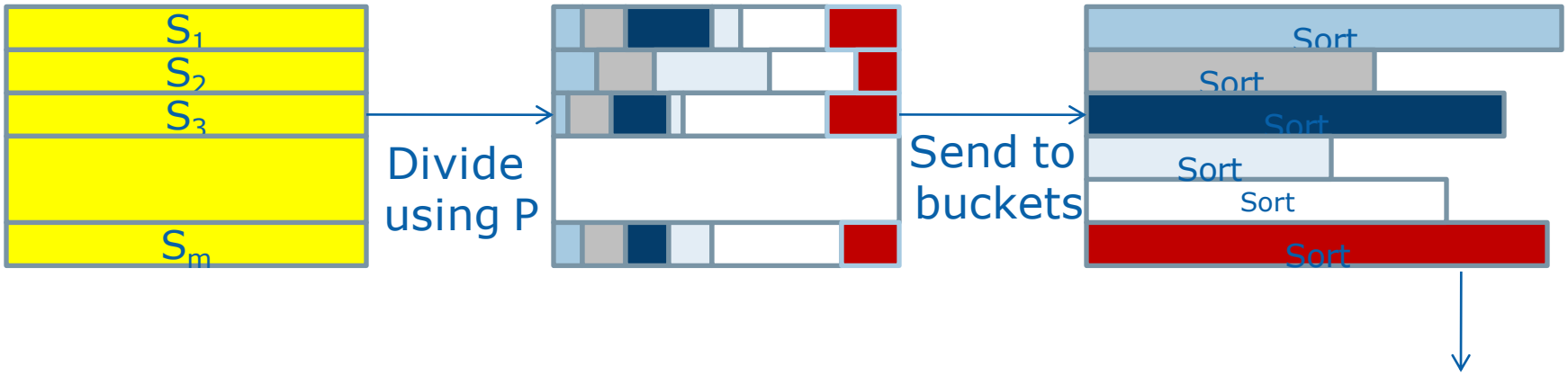


- We know segment lengths & offsets - Can transfer in parallel
- Each of the chunks might be very small compared to B. Cache miss overhead is high for naïve copying.
- Our approach: D&C on the <row, bucket> index space
- Invoke BKT-TRANSPOSE(1,1,$n^{1/2}$) [Q(n;M,B) = only O(n/B)]
- BKT-TRANSPOSE (x_0, y_0, size)

  IF size =1, just copy the one segment

  ELSE {BKT-TRANSPOSE (x_0 , y_0 , size/2); BKT-TRANSPOSE (x_0 + size/2, y_0 , size/2); two more recursive calls}

Sorting complexity



8. Finally, sort buckets

ANSWER

- $Q(n;M,B) = n^{1/2}Q(n^{1/2};M,B)+\sum Q(x_i;M,B) + O(n/B)$,

  $= O((n/B)(\log_{(M/B)}(n/B))$.

- Work(n) = O(n log n). Depth(n) = $O(\log^2(n))$
- Depth of randomized version: $O(\log^{3/2}(n))$
- [Cole, Ramachandran '10] improves depth of deterministic sort to O(logn loglog n)

Sorting leads to graph algorithms

Problem	Depth	Cache complexity
List ranking	$O(D_{sort}(n) \log n)$	$O(Q_{sort}(n))$
Euler tour on trees (Successor + L.R.)	$O(D_{LR}(n))$	$O(Q_{sort}(n))$
Tree contraction (Euler Tour + Indep. Set)	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(n))$
Least common ancestors (k queries) (Euler Tour + Range Minima Query)	$O(D_{LR}(n))$	$O(\lceil k/n \rceil Q_{sort}(n))$
Connected components (Tree Contraction)	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(\|E\|) \log(\|V\|/\sqrt{M}))$
MST (Tree Contraction)	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(\|E\|) \log(\|V\|/\sqrt{M}))$

BT-Sort: Experimental Study

	weight	STL Sort	Sanders Sort	Quicksort	BT-Sort	BT-Sort
Cores		1	32	32	32	1
Uniform	.1	15.8	1.06	4.22	.82	20.2
Exponential	.1	10.8	.79	2.49	.53	13.8
Almost Sorted	.1	3.28	1.11	1.76	.27	5.67
Trigram Strings	.2	58.2	4.63	8.6	1.05	30.8
Strings Permuted	.2	82.5	7.08	28.4	1.76	49.3
Structure	.3	17.6	2.03	6.73	1.18	26.7
Average		**36.4**	**3.24**	**10.3**	**1.08**	**28.0**

- **Time in seconds on 32 core Nehalem (4 X x7560)**
- **All inputs are 100,000,000 long**
- **All code written run on Cilk++ (also tested in Cilk+)**
- **BT-Sort follows Low-Span + Cache-Oblivious approach**

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?**

NEXT UP

Lecture #2: Tackling Multi-level Hierarchies

References

[Acar, Blelloch, Blumofe '02] Umut A. Acar, Guy E. Blelloch, Robert D. Blumofe. The Data Locality of Work Stealing. Theory Comput. Syst. 35:3 (2002)

[Blelloch, G '04] G. E. Blelloch and P. B. Gibbons. Effectively Sharing a Cache Among Threads. ACM SPAA, 2004

[Blelloch, G, Matias '99] G. E. Blelloch, P. B. Gibbons and Y. Matias. Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. J. ACM 46:2 (1999)

[Blelloch, G, Simhadri '10] G. E. Blelloch, P. B. Gibbons and H. V. Simhadri. Low Depth Cache-Oblivious Algorithms. ACM SPAA, 2010

[Blumofe, Leiserson '99] Robert D. Blumofe, Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. J. ACM 46:5 (1999)

[Blumofe et al. '96] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, Keith H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. ACM SPAA, 1996

[Brent '74] Richard P. Brent. The parallel evaluation of general arithmetic expressions" J. ACM 21:2, 1974

[Cole, Ramachandran '10] Richard Cole, Vijaya Ramachandran. Resource Oblivious Sorting on Multicores. ICALP, 2010

[Fortune, Wyllie '78] Steven Fortune, James Wyllie. Parallelism in Random Access Machines. ACM STOC, 1978

[Frigo, Leiserson, Prokop, Ramachandran '99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, Sridhar Ramachandran. Cache-Oblivious Algorithms. IEEE FOCS 1999

[Shiloach, Vishkin '82] Yossi Shiloach, Uzi Vishkin. An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm. J. Algorithms 3:2 (1982)