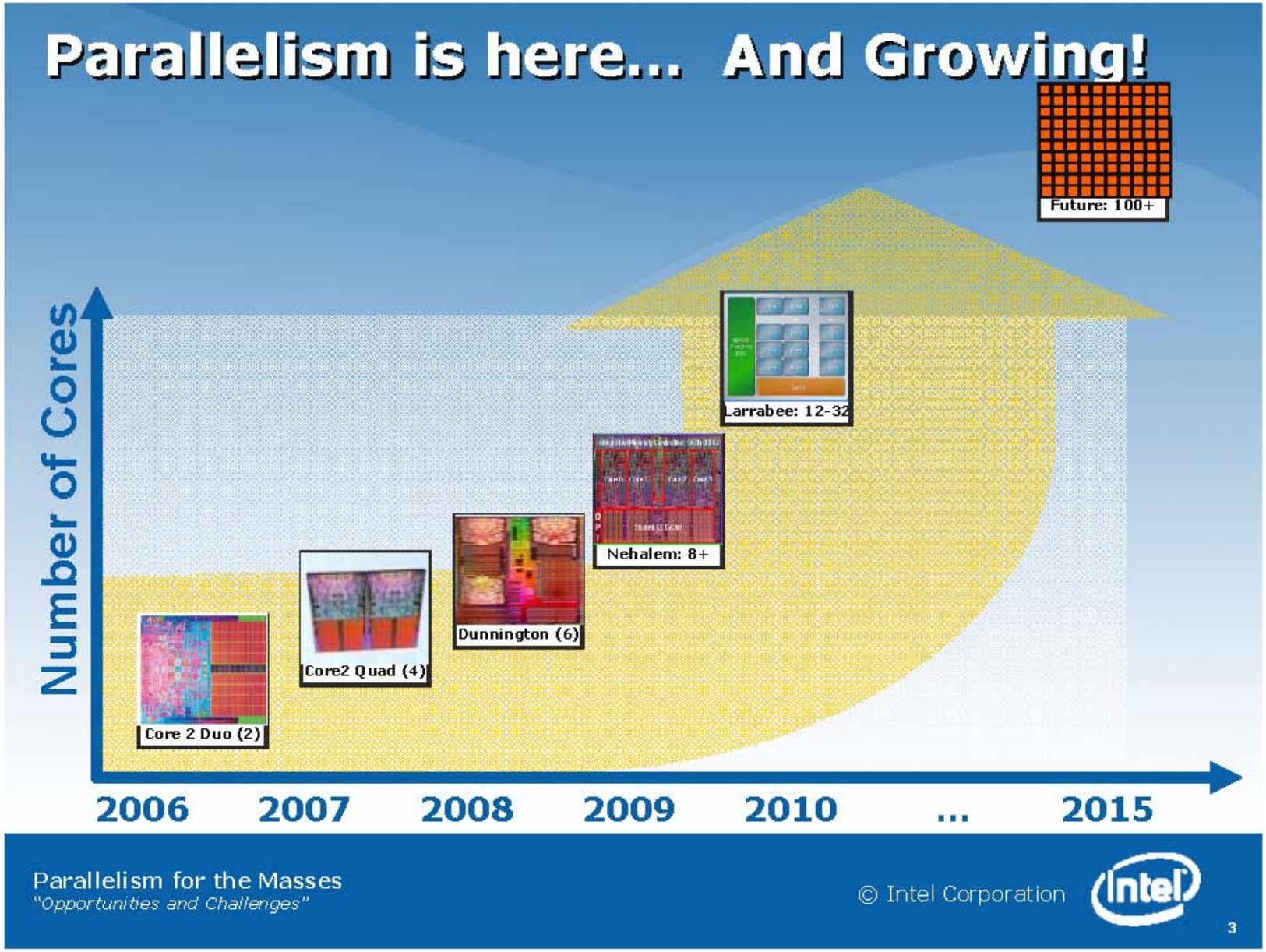
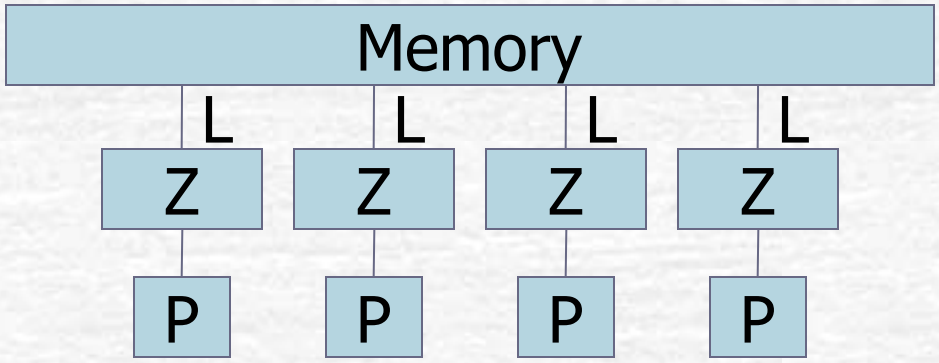
Algorithms for Parallel Cache Hierarchies

Guy Blelloch
Carnegie Mellon University

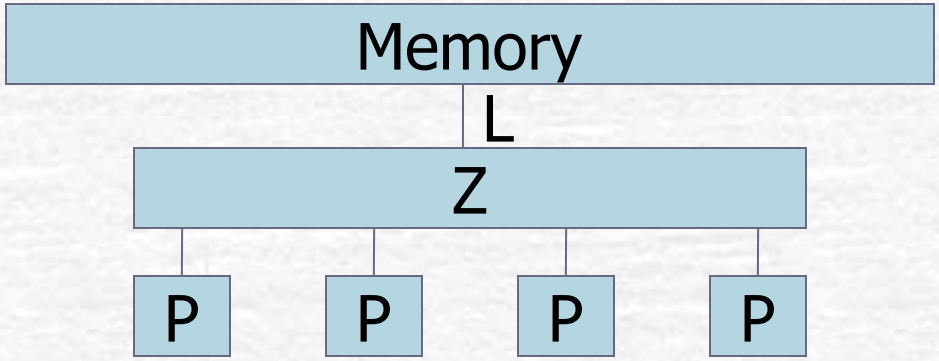


Andrew Chien, 2008

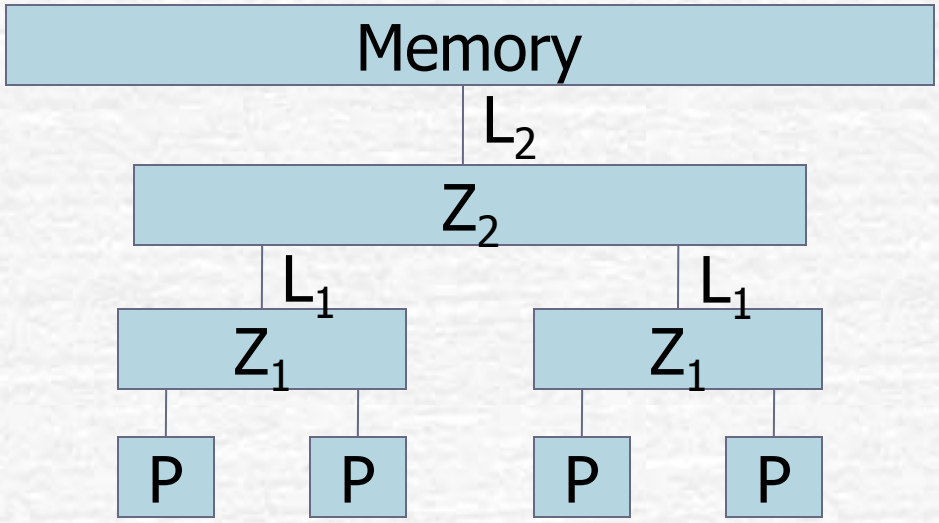
Memory Hierarchies



Private cache



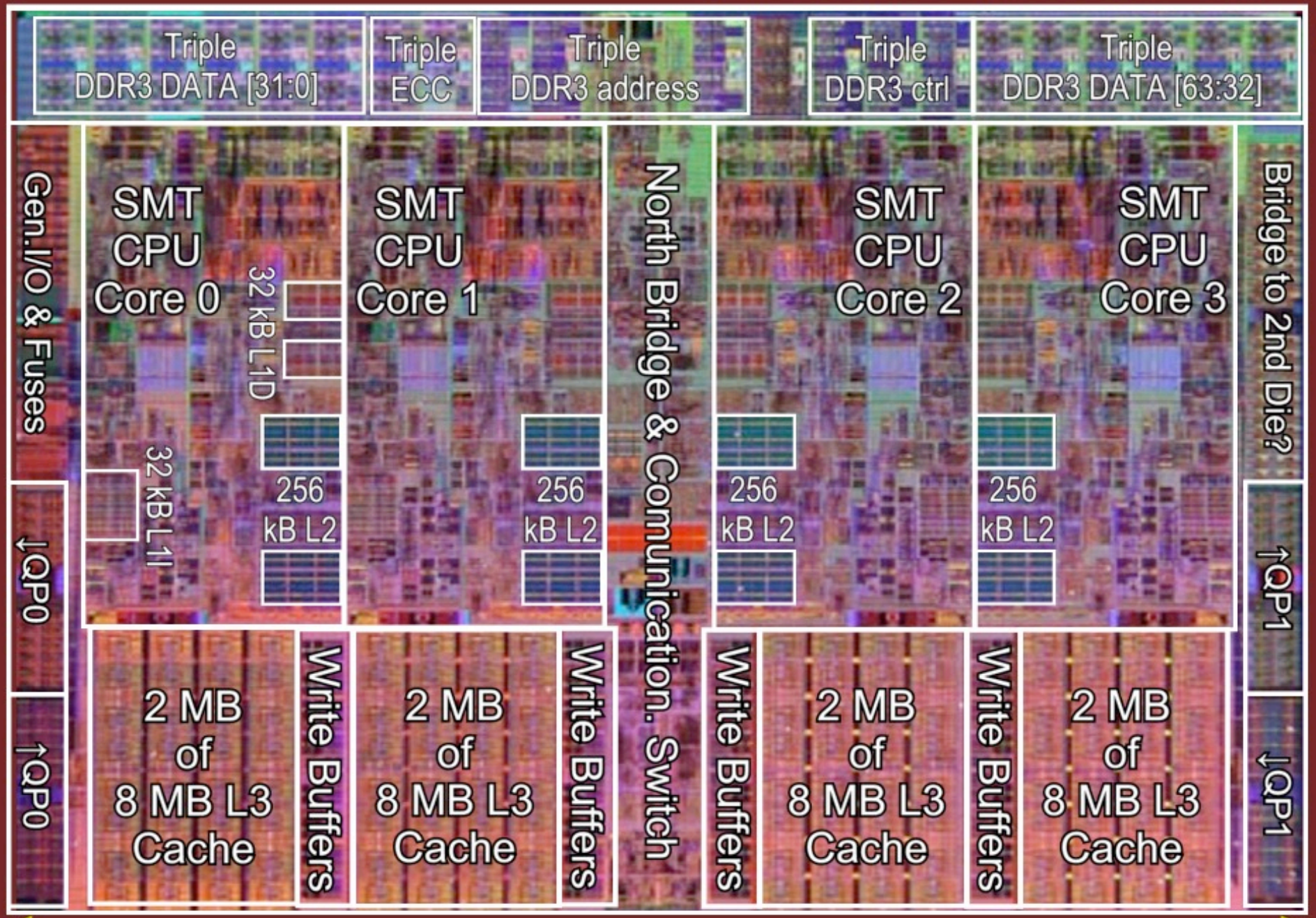
Shared cache



Hierarchical cache

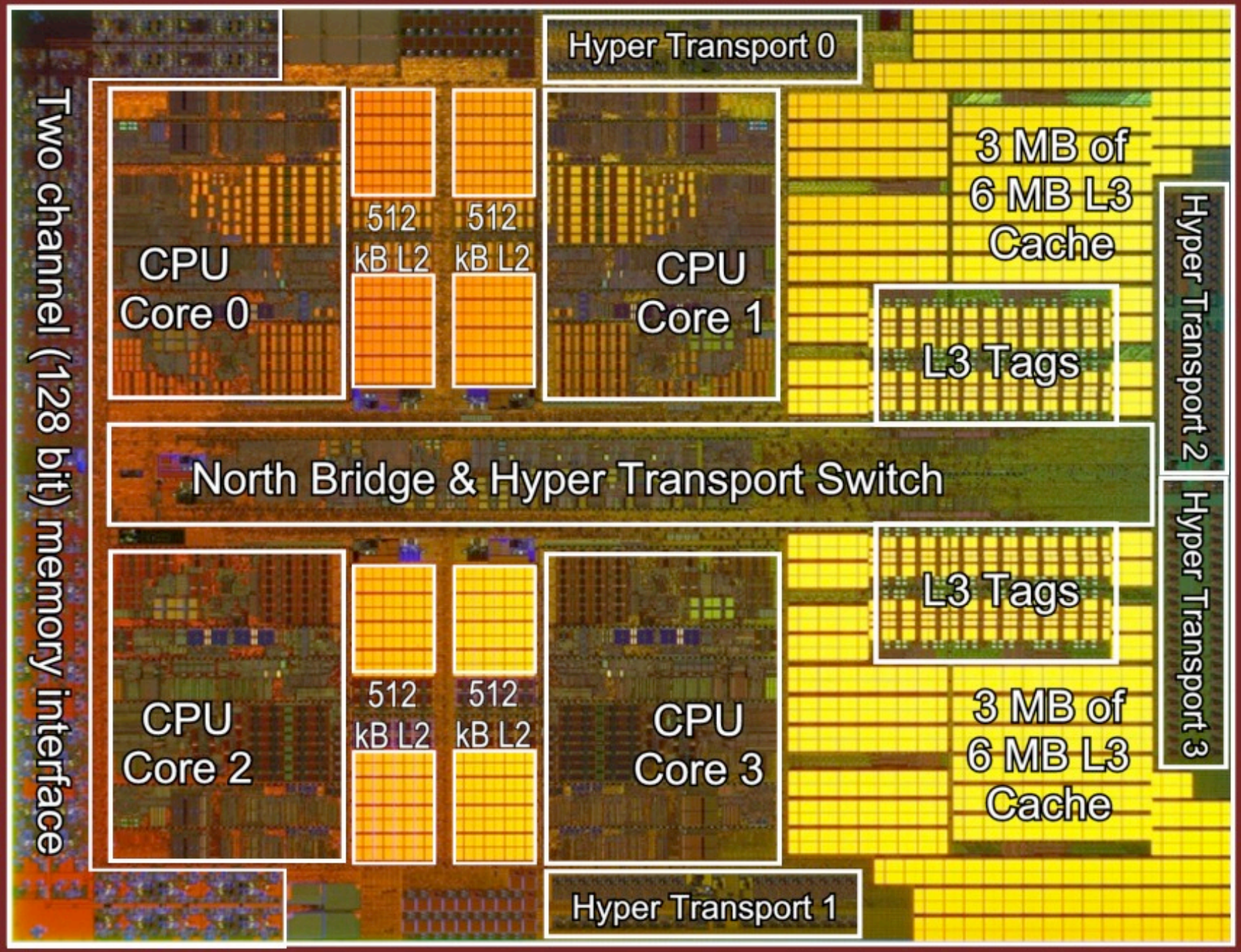
L2 cache tiles: 7.1 mm2 / MB; L3 cache tiles: 5.7 mm2 / MB (excl.tags)

Die size 246 mm2 (incl. test circ.265 mm2)

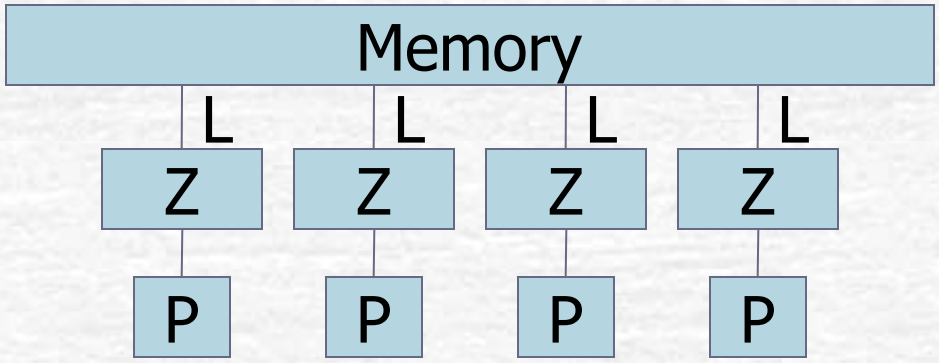


L2 cache tiles: 7.5 mm2 / MB, L3 cache tiles: 7.5 mm2 / MB (excl.tags)

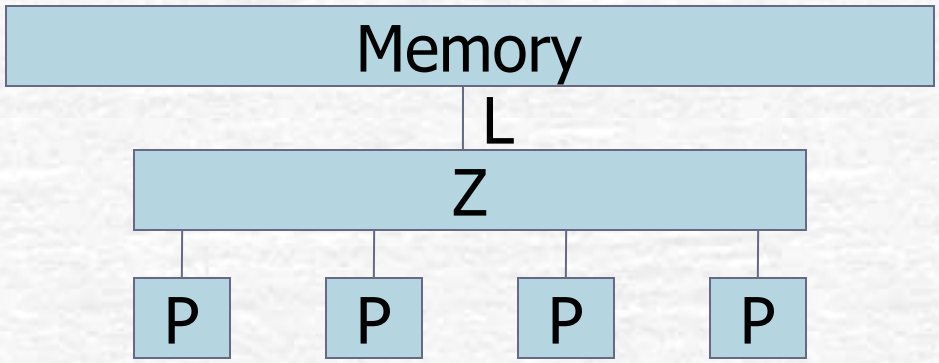
Die size 243 mm2 (incl. test circ.263 mm2)



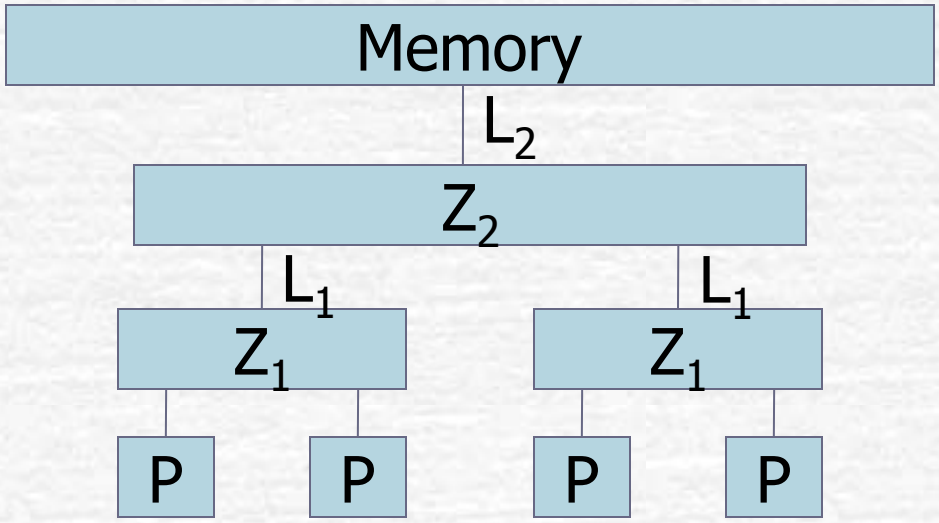
What about in Parallel



Private cache



Shared cache



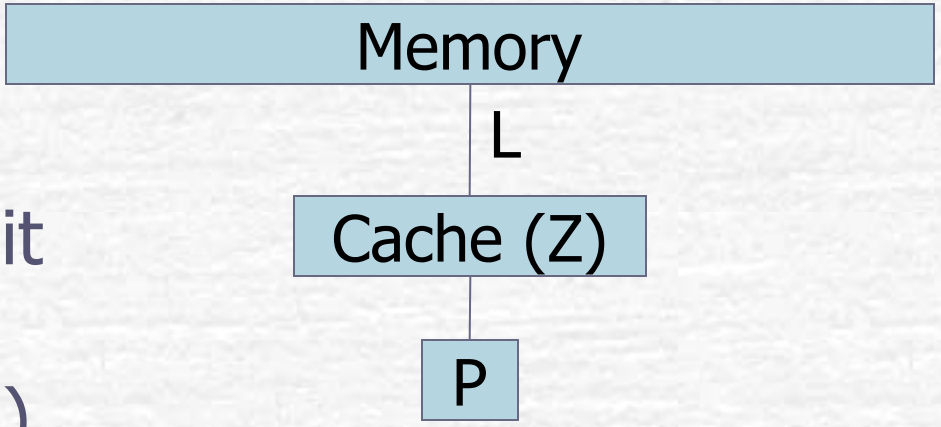
Hierarchical cache
(e.g. Valiant, 08)

Observation

- Many “parallel” algorithms have natural “sequential” locality
- Can we take advantage of this on a parallel machine. In particular can we describe/ program/analyze such algorithms in a high-level way, but still understand performance on various cache architectures.

Sequential Locality

- Assume unbounded memory and an ideal cache with a capacity of Z words and cache-lines of L words each.
  - Q(C; Z, L) – number of cache misses
- An algorithm is **cache-oblivious** if it does not use these parameters (Z and L)



Quicksort

```
function quicksort(S) =
if (#S <= 1) then S
else let
  a = S[rand(#S)];
  S1 = {e in S | e < a};
  S2 = {e in S | e = a};
  S3 = {e in S | e > a};
  R = {quicksort(v) : v in [S1, S3]};
in R[0] ++ S2 ++ R[1];
```

NESL code

Cache Oblivious with: Q(n;Z,L) = O(n/L log n/Z) w.h.p

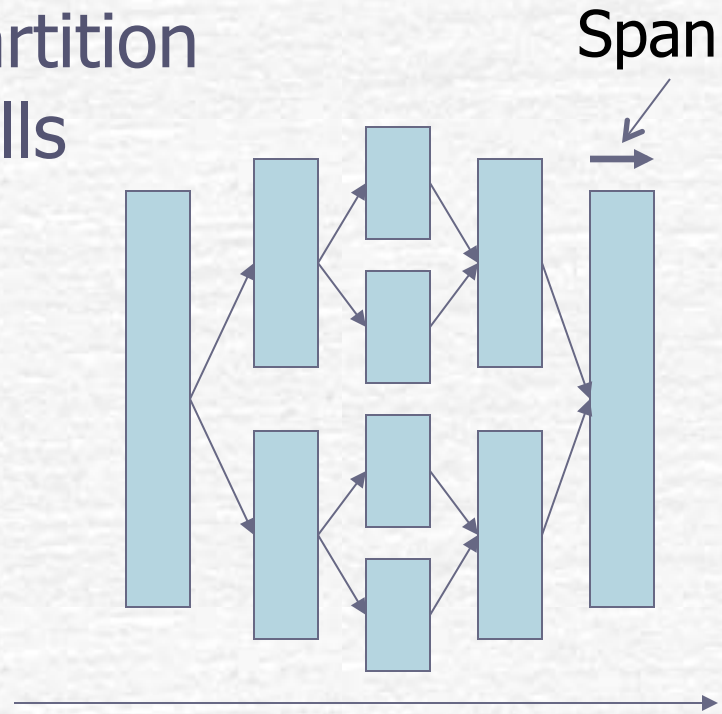
{} indicates parallelism

Nested parallelism

- Standard programming model with fork-join parallelism and no synchronization among tasks.
  - No notion of processors (processor oblivious).
  - Cost calculated using:
    - Work (W) : sum over parallel calls
    - Span (D) : take maximum over parallel calls
  - ID, NESL, Cilk++, X10, Open MP, Microsoft TPL
  - Typically much more parallelism than processors
  - **Lots of flexibility for scheduler**

Qsort Complexity

Parallel partition
Parallel calls



Work = O(n log n)

A good parallel algorithm

Parallelism = O(n/log n)

Greedy Schedules

Greedy schedule: a processor cannot sit idle if there is work to do:

For any greedy schedule ([EZL, 1989]):

$$\max\left(\frac{W}{P}, D\right) \leq T_P \leq \frac{W}{P} + D$$

What about:

- Space usage
- Scheduling overheads
- **Locality**

Matrix Multiplication

```
Fun A*B {
  if #A < k then baseCase..
  C11 = A11*B11 + A12*B21
  C12 = A11*B12 + A12*B22
  C21 = A21*B11 + A22*B21
  C22 = A21*B12 + A22*B22
  return C
}
```

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} W_*(n) &= 8W(n/2) + O(n^2) \\ &= O(n^3) \end{aligned}$$

$$\begin{aligned} D(n) &= D(n/2) + O(1) \\ &= O(\log n) \end{aligned}$$

$$Parallelism = \frac{W}{D} = O\left(\frac{n^3}{\log n}\right)$$

$$Q(n;Z,L) = O\left(\frac{n^{1.5}}{LZ^{.5}}\right)$$

Matrix Inversion

```
fun invert(M) {
  if small baseCase
  D^-1 = invert(D)
  S  = A - BD^-1C
  S^-1 = invert(S)
  E = S^-1
  F = S^-1BD^-1
  G = -D^-1CS^-1
  H = D^-1 + D^-1CS^-1BD^-1}
```

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

$$M^{-1} = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$W(n) = 2W(n/2) + 6W_*(n/2) = O(n^3)$$

$$D(n) = 2D(n/2) + 6D_*(n/2) = O(n)$$

$$Parallelism = \frac{W}{D} = O\left(n^2\right)$$

Summary of Results

$$Q(n;Z,L) =$$

Scan Memory, prefix sums, merge, median, matrix transpose: $O\left(\frac{n}{L}\right)$

Matrix Multiply $O\left(\frac{n^{1.5}}{LZ^{.5}}\right)$

Matrix Inversion:

FFT: $O\left(\frac{n}{L}\log_Z n\right)$

Mergesort, Quicksort, NNs, KD-trees: $O\left(\frac{n}{L}\log_2(n/Z)\right)$

Funnel Sort: $O\left(\frac{n}{L}\log_Z n\right)$

Summary of Results

$$D(n) =$$

Scan Memory, prefix sums, merge, median, matrix transpose: $O(\log n)$

Matrix Multiply $O(\log n)$

Matrix Inversion: $O(\sqrt{n})$

FFT: $O(\log^2 n)$

Mergesort, Quicksort, NNs, KD-trees: $O(\log^2 n)$

~~Funnel Sort:~~ Blocked Sample Sort: $O(\log^2 n)$

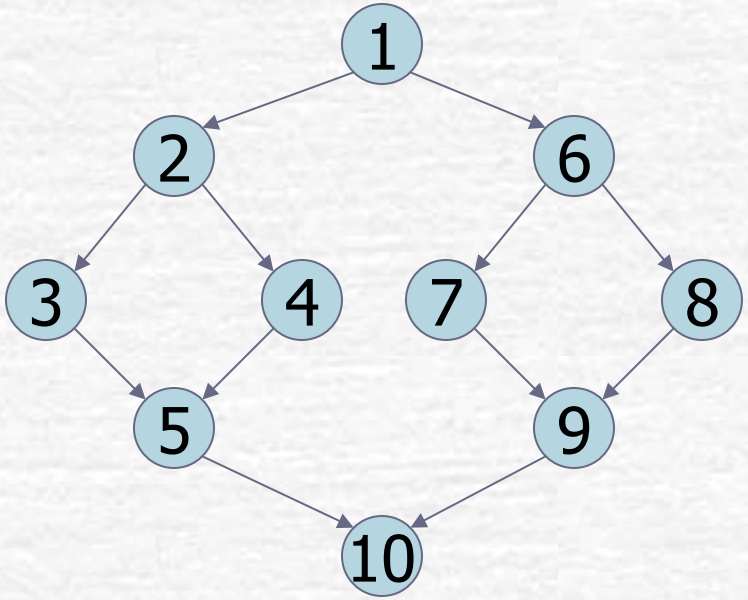
Some Basic Results

- Scan memory: $D(n) = O(\log n)$
- Matrix transpose (divide-and-conquer):
  $D(n) = O(\log n)$
- Simple matrix multiply (divide-and-conquer):
  $D(n) = O(\log^2 n)$
- Quicksort and Mergesort:
  $D(n) = O(\log^2 n)$
- ~~Funnel sort~~: $D(n) = O(n)$

Depth-first (sequential) schedule

Depth-First ordering.

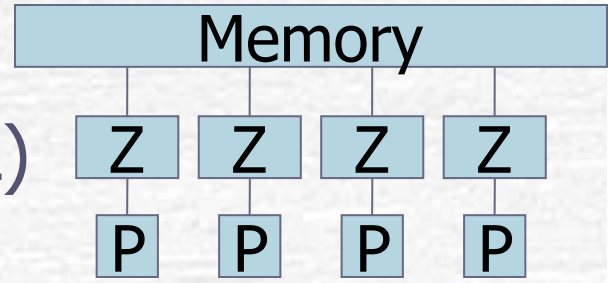
Same as sequential execution



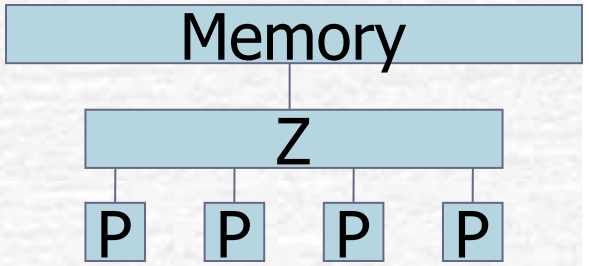
$Q_1(C;Z,L)$ – cache complexity for DFS order.

Greedy Schedulers

- Work stealing ([B+96,ABB00])
  $Q_P(C;Z,L) = Q_1(C;Z,L) + O(PDZ/L)$
  private cache



- P-DFS [BG02]
  $Q_P(C;Z+PDL,L) = Q_1(C;Z,L)$
  shared cache

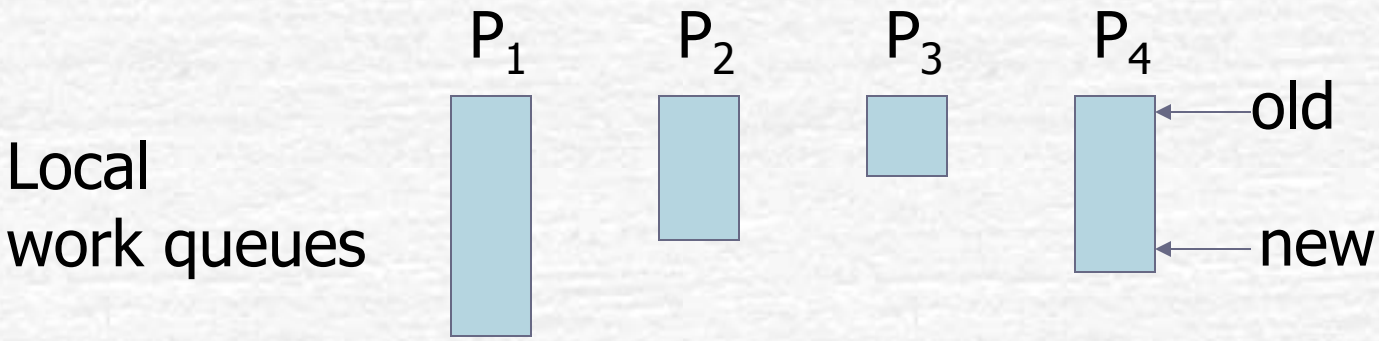


Eg. Matrix multiply

$Q_p(n;Z,L) = O(n^{3/2}/LZ^{1/2} + PZ \log n/L)$

$Q_P(n;Z+PL \log n,L) = O(n^{3/2}/LZ^{1/2})$

Work Stealing

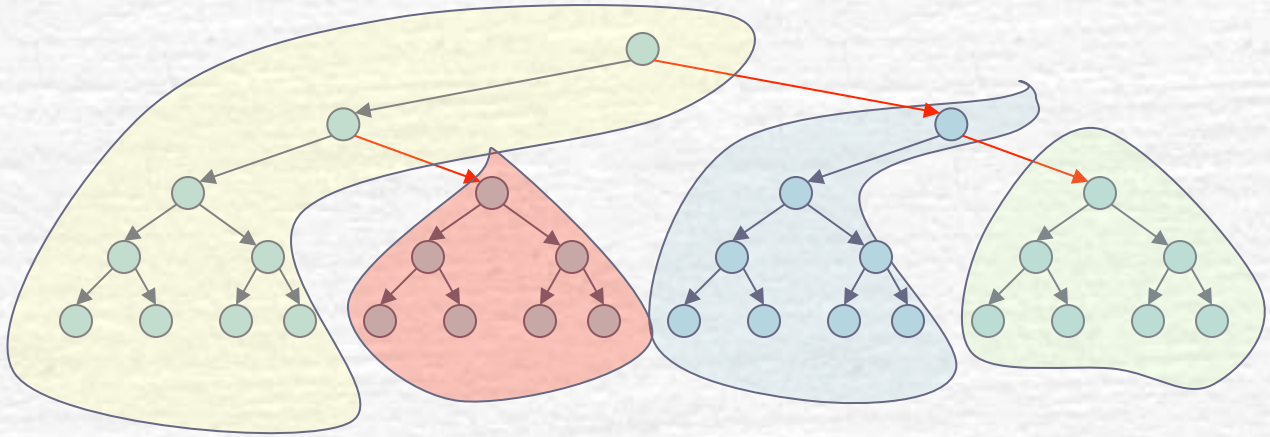


- push new jobs on "new" end
- pop jobs from "new" end
- If processor runs out of work, then "**steal**" from another "old" end

Each processor tends to execute a sequential part of the computation.

Work Stealing

Tends to schedule "sequential blocks" of tasks

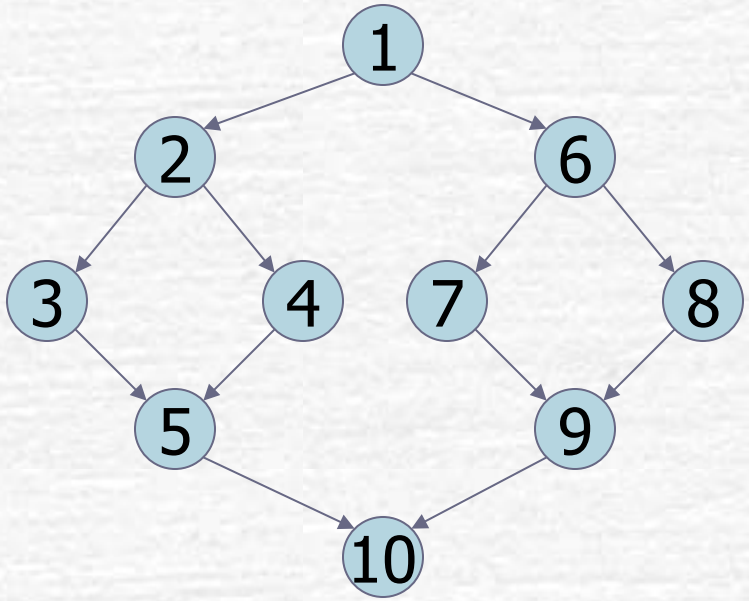


= steal

#steals = O(PD) [BL98]

Parallel Depth First Schedules (P-DFS)

List scheduling based on Depth-First ordering



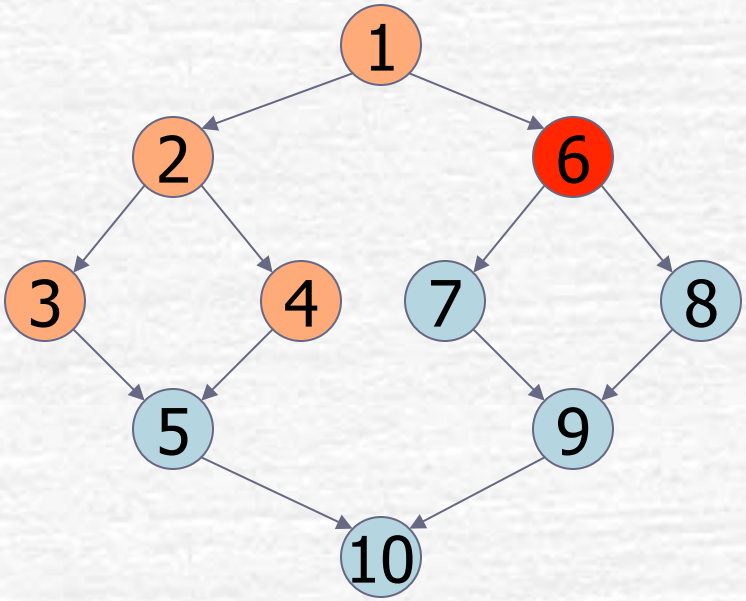
2 processor schedule

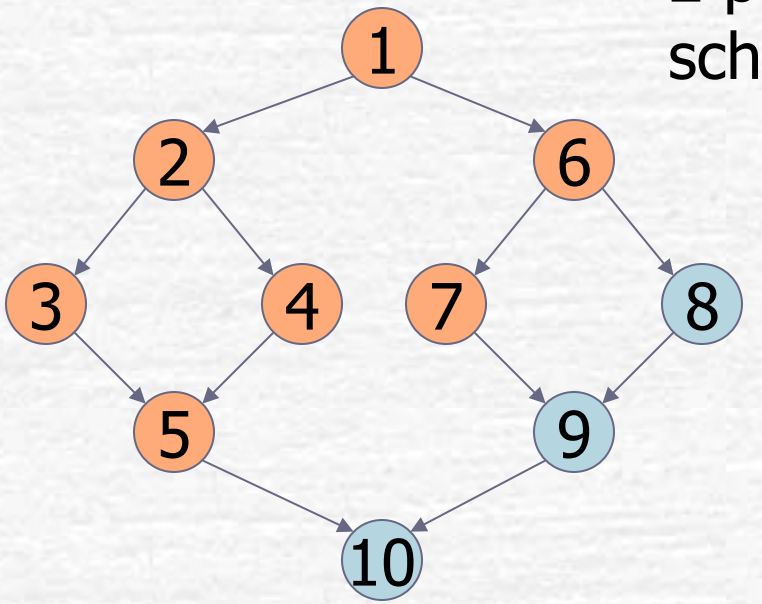
1
2, 6
3, 4
5, 7
8
9
10

For strict computations a shared stack implements a P-DFS

"Premature task" in P-DFS

A running task is premature if there is an earlier sequential task that is not complete





2 processor schedule

1
2, 6
3, 4
5, 7
8
9
10

= premature

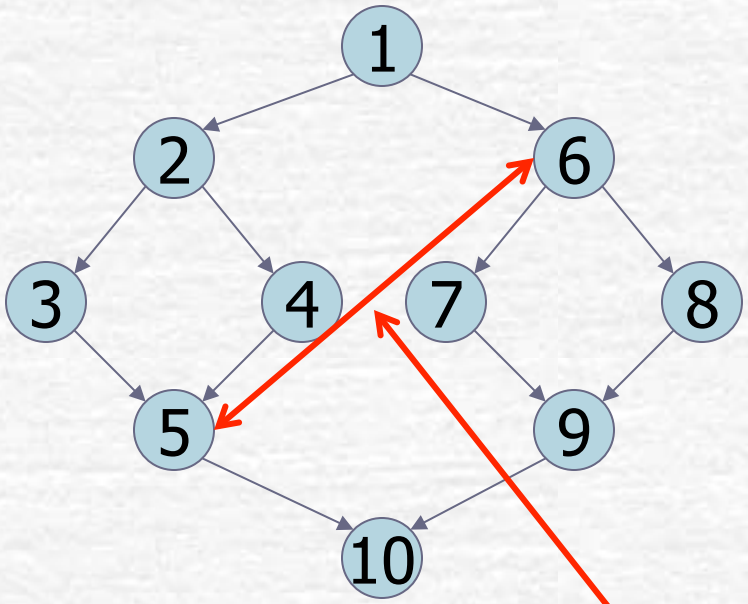
#premature nodes = O(PD)

What problems remain?

- What about sorting?
  $O(\log^2 n)$ span version of sample sort [BGV09]
- Other algorithms? (e.g. geometry, dynamic data structures)
- What about high-span algorithms
  Work-efficient Matrix inversion has span $O(n^{1/2})$
- What about hierarchical caches???
  - Idea 1: modify definition of Q_P
  - Idea 2: balance space and work

Depth-first (sequential) schedule

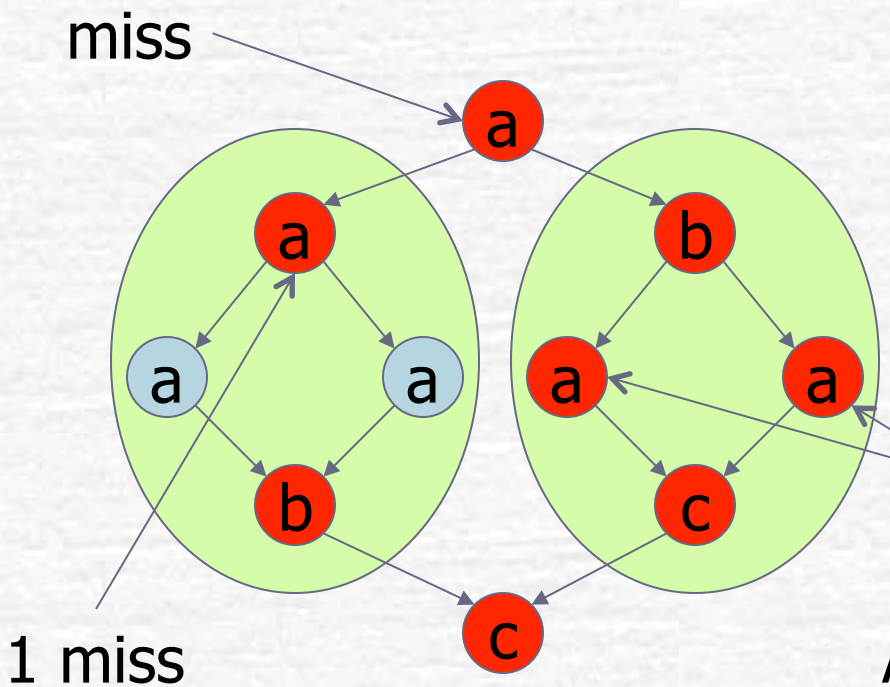
Depth-First ordering.



$Q_1(C;Z,L)$ – cache complexity for DFS order.

Captures artificial locality

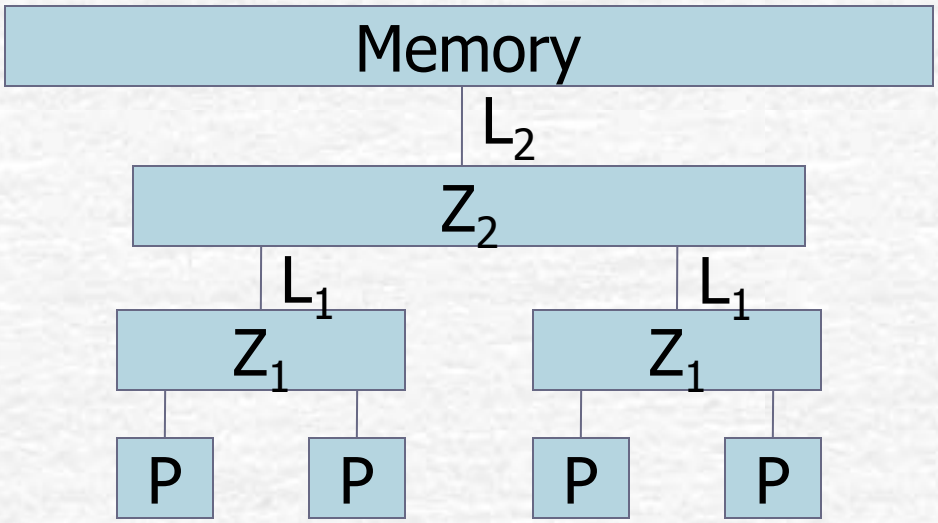
Idea 1: A variant of the CO model

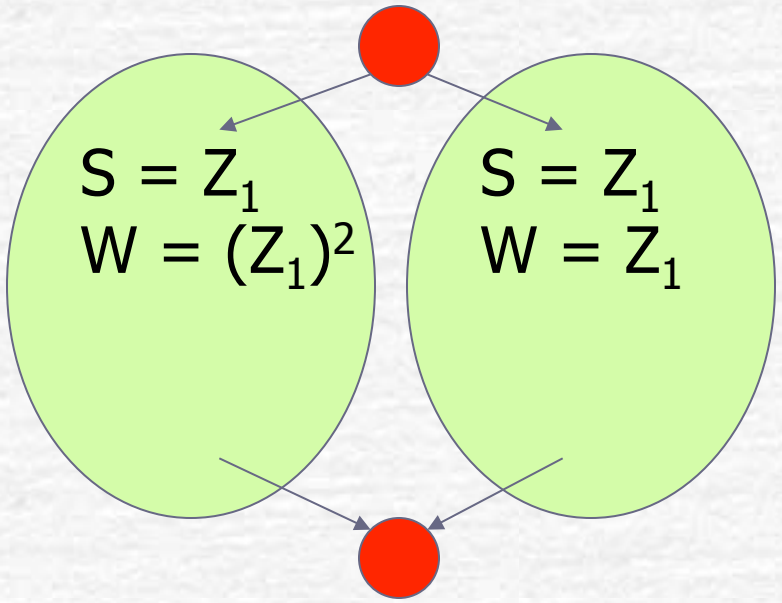


Idea 1: If a task fits in cache, then any ordered access to the same location will be a cache hit, otherwise an access is a miss

All algorithms we have studied are not asymptotically affected

Idea 2





- Note that processing power is tied to space. We need to somehow balance space and work.

Idea 2: Effective Cache Complexity

$$Q^*(a \,\|\, b) = \max\begin{cases} Q^*(a) + Q^*(b) \\ s(a \,\|\, b)^{\alpha} \times \max\left\{ \dfrac{Q^*(a)}{s(a)^{\alpha}}, \dfrac{Q^*(b)}{s(b)^{\alpha}} \right\} \end{cases}$$

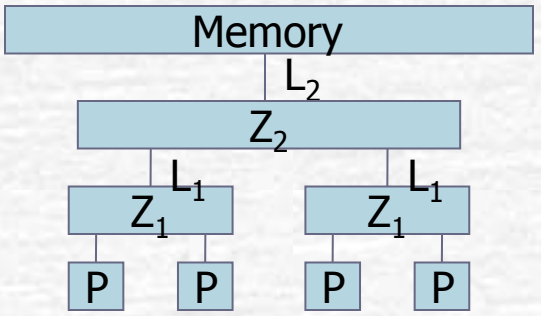
Where s(x) is the space taken by x.

Parallelism is bounded by $s(a)^{\alpha}$

$\alpha = 0$, is sequential execution.

For all algorithms we considered top term dominates when $\alpha < 1$

Main Result



A scheduler such that:

$$T(n) = \sum_{i=0}^{l} C_i Q^*(n; Z_i, L_i) / P$$

For an l-level cache: (C_0,Z_0,L_0,P_0), … , (C_l,Z_l,L_l,P_l)

$P_0 = Z_0 = L_0 = 1$ (operations on registers)

$P = P_1 \times \ldots \times P_l$

With condition (approx): $P_i < (Z_i/Z_{(i-1)})^{\alpha}$

*Requires space annotations on tasks

Conclusions

1. Can model locality at a high level
2. Many cache-oblivious algorithms are naturally parallel, and cache-oblivious nature can be used on hierarchical caches.
3. Many open problems.