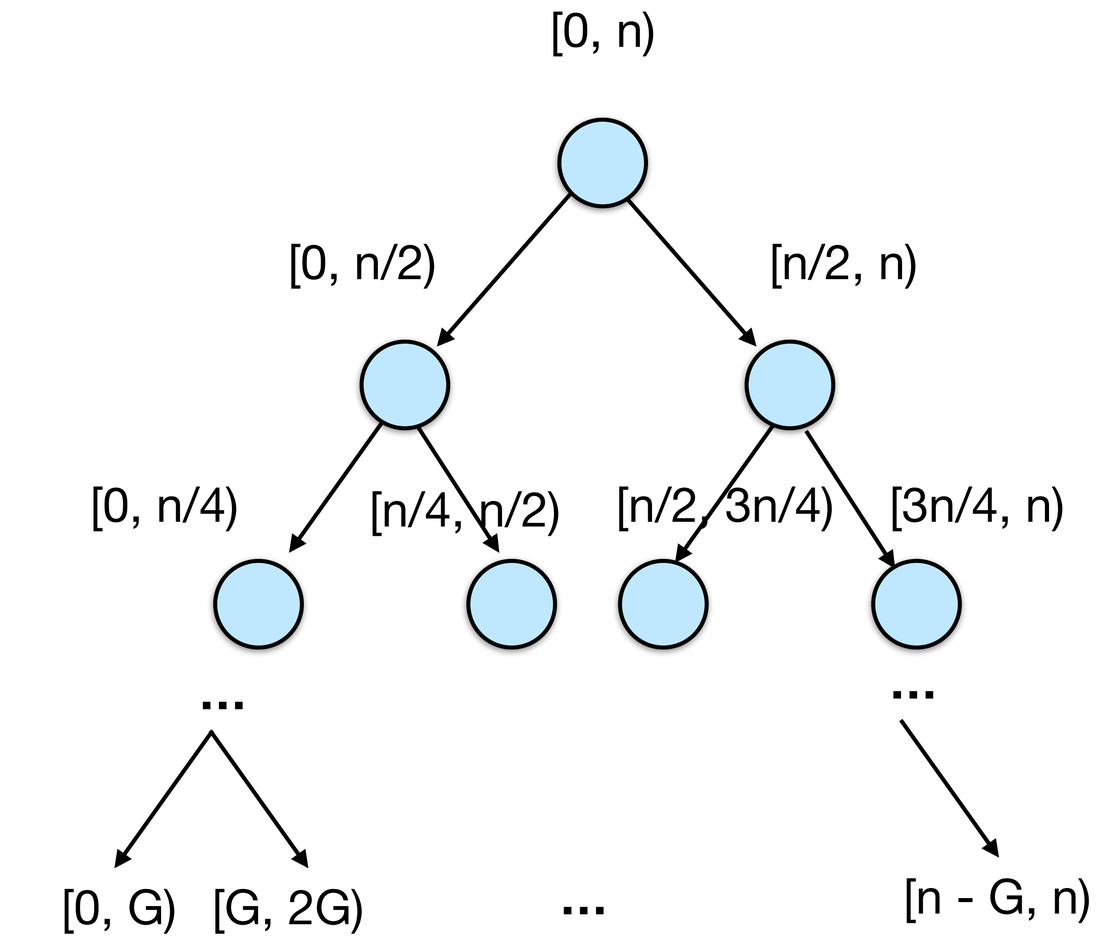
Parallel loops

Coarsening for parallel overhead

G = $\Theta(1)$

```
void p_loop(int lo, int hi) {
  if (hi > lo + G) {
    int mid = lo + (hi - lo) / 2;
    parallel_spawn p_loop(lo, mid);
    p_loop(mid, hi);
    parallel_sync;
    return;
  }
  for (int i = lo; i < hi; ++i) {
    // loop body
  }
}
```

$\Theta(\lg n)$



Merge Sort
(resuming where we left off last time)

Merging Two Sorted Arrays

```
void merge(int *C, int *A, int na, int *B, int nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++, na--;
    } else {
      *C++ = *B++, nb–;
    }
  }
  while (na > 0) {
    *C++ = *A++, na--;
  }
  while (nb > 0) {
    *C++ = *B++, nb–;
  }
}
```

Work to merge n elements = $\Theta(n)$

Merge Sort

```
void merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

Work of Merge Sort

```
void merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \Theta(n^{\log_b a} \lg^0 n)$$

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$$= \Theta(n \lg n)$$

From MIT 6.172

Span of Merge Sort

```
void merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Case 3 of Master Method

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = \Theta(n)$$

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(n)$

$= \Theta(n)$

Parallelism of Merge Sort

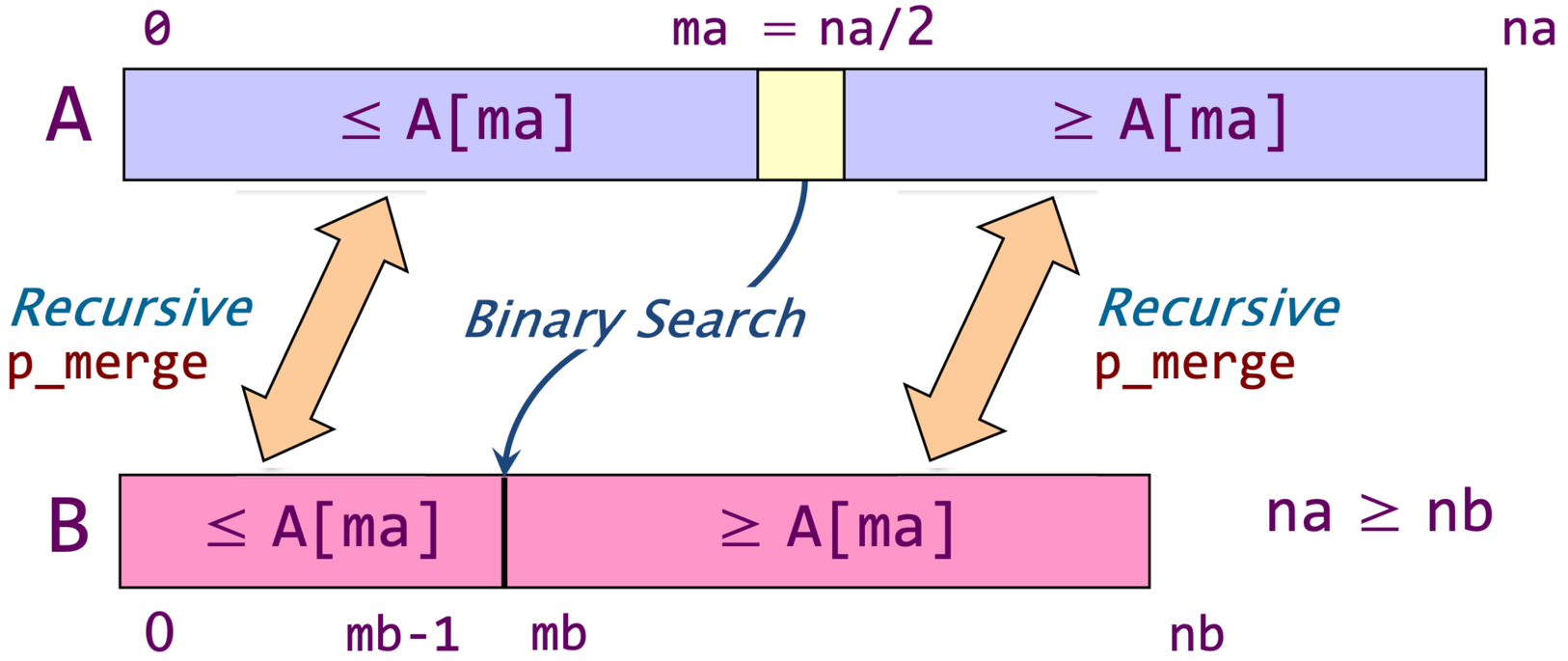
Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(n)$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(\lg n)$

We need to parallelize the merge!

Parallel Merge



KEY IDEA: If the total number of elements to be merged in the two arrays is $n = na + nb$, the total number of elements in the larger of the two recursive merges is at most $(3/4)n$.

Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
  if (na < nb) {
    p_merge(C, B, nb, A, na); // flip A and B
  } else if (na == 0) {
    return;
  } else {
    int ma = na/2;
    int mb = binary_search(A[ma], B, nb);
    C[ma+mb] = A[ma];
    parallel_spawn p_merge(C, A, ma, B, mb);
    p_merge(C+ma+mb+1, A+ma+1, na-ma-1, B+mb, nb-mb);
    parallel_sync;
  }
}
```

Coarsen base cases for efficiency.

From MIT 6.172

Span of Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
  if (na < nb) {
    p_merge(C, B, nb, A, na); // flip A and B
  } else if (na == 0) {
    return;
  } else {
    int ma = na/2;
    int mb = binary_search(A[ma], B, nb);
    C[ma+mb] = A[ma];
    parallel_spawn p_merge(C, A, ma, B, mb);
    p_merge(C+ma+mb+1, A+ma+1, na-ma-1, B+mb, nb-mb);
    parallel_sync;
  }
}
```

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_{4/3} 1} = 1$$

$$f(n) = \Theta(n^{\log_b a} \lg^1 n)$$

Span: $T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)$

$= \Theta(\lg^2 n)$

From MIT 6.172

Work of Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
  if (na < nb) {
    p_merge(C, B, nb, A, na); // flip A and B
  } else if (na == 0) {
    return;
  } else {
    int ma = na/2;
    int mb = binary_search(A[ma], B, nb);
    C[ma+mb] = A[ma];
    parallel_spawn p_merge(C, A, ma, B, mb);
    p_merge(C+ma+mb+1, A+ma+1, na-ma-1, B+mb, nb-mb);
    parallel_sync;
  }
}
```

Complicated / not in Master Method cases

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,

where $1/4 \leq \alpha \leq 3/4$

Claim: $T_1(n) = \Theta(n)$

Analysis of Work Recurrence

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n),$$
where $1/4 \le \alpha \le 3/4$.

Substitution method: Inductive hypothesis is $T_1(k) \le c_1 k - c_2 \lg k$, where $c_1, c_2 > 0$. Prove that the relation holds, and solve for c_1 and c_2.

$$\begin{aligned}
T_1(n) &\le c_1(\alpha n) - c_2\lg(\alpha n) + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1 n - c_2\lg(\alpha n) - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1 n - c_2\,(\lg(\alpha(1-\alpha)) + 2\lg n\,) + \Theta(\lg n) \\
&\le c_1 n - c_2\lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\le c_1 n - c_2\lg n,
\end{aligned}$$

by choosing c_2 large enough. Choose c_1 large enough to handle the base case.

Parallelism of Parallel Merge

Work: $T_1(n) = \Theta(n)$

Span: $T_\infty(n) = \Theta(\lg^2 n)$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n/\lg^2 n)$

Parallel Merge Sort

```
void p_merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    p_merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Same as before except with parallel merge

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \Theta(n^{\log_b a} \lg^0 n)$$

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

Parallel Merge Sort

```
void p_merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    p_merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Same as before except with parallel merge

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = \Theta(n^{\log_b a} \lg^2 n)$$

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$

$= \Theta(\lg^3 n)$

Parallelism of Parallel Merge Sort

Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(\lg^3 n)$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n/\lg^2 n)$

Tableau Construction

Constructing a Tableau

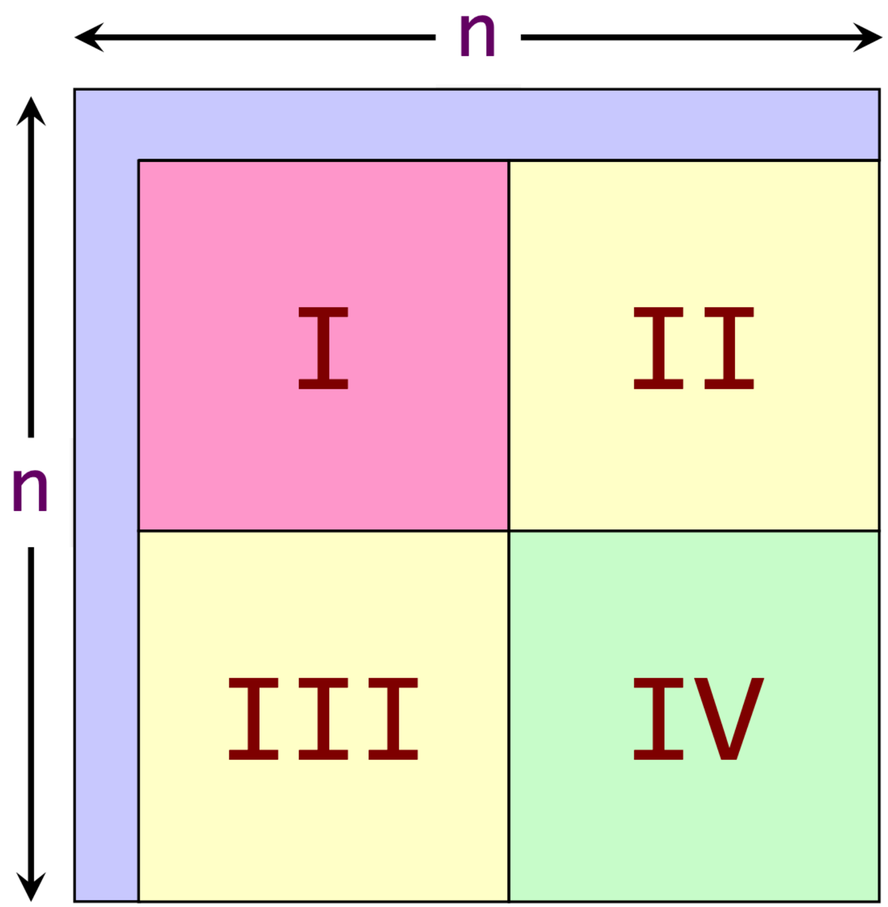
Problem: Fill in an n x n tableau A, where A[i][j] = f(A[i][j–1], A[i–1][j], A[i–1][j–1]).

00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Dynamic programming
- Longest common subsequence
- Edit distance
- Time warping

Work = $\Theta(n^2)$

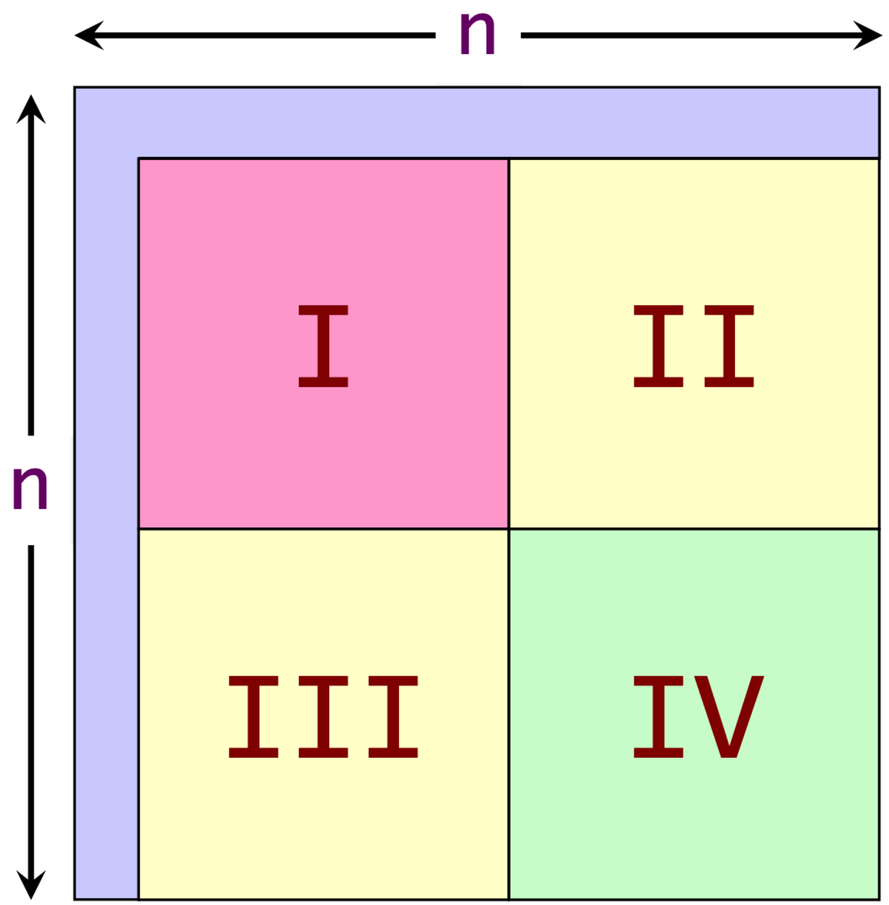
Recursive Construction




```
I;
parallel_spawn II;
III;
parallel_sync;
IV;
```

What is the recurrence for the work?

Recursive Construction




```
I;
parallel_spawn II;
III;
parallel_sync;
IV;
```

Work: $T_1(n) = 4T_1(n/2) + \Theta(1) = \Theta(n^2)$

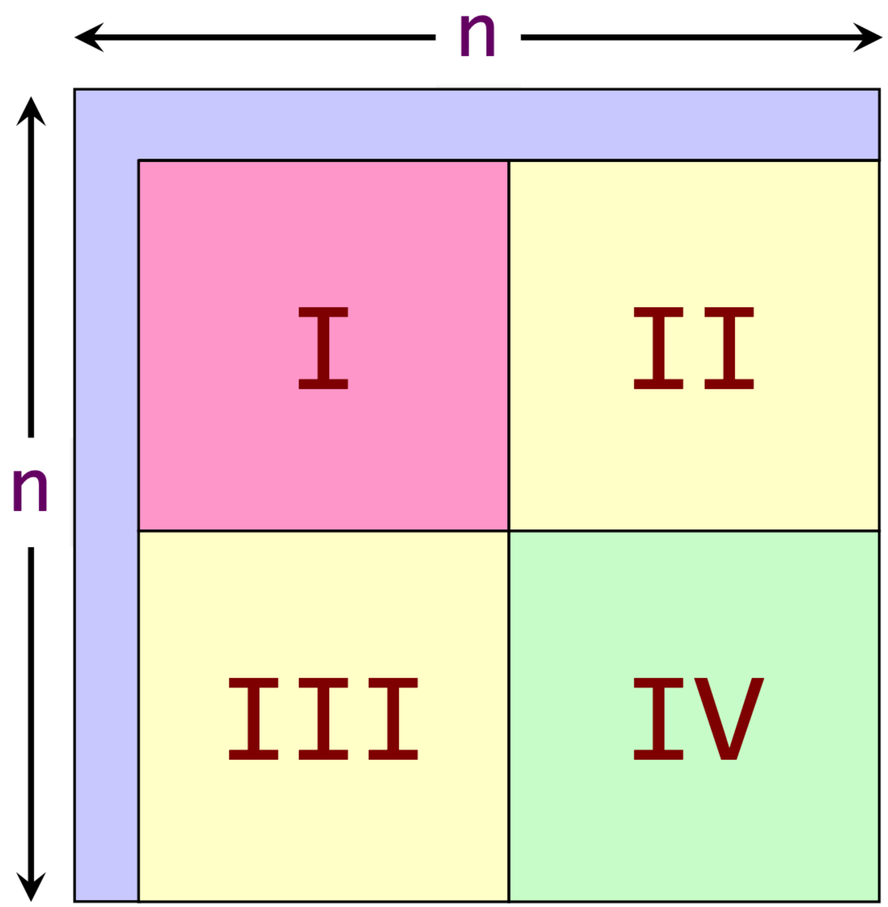
Case 1 of Master Method

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

$$f(n) = \Theta(1)$$

From MIT 6.172

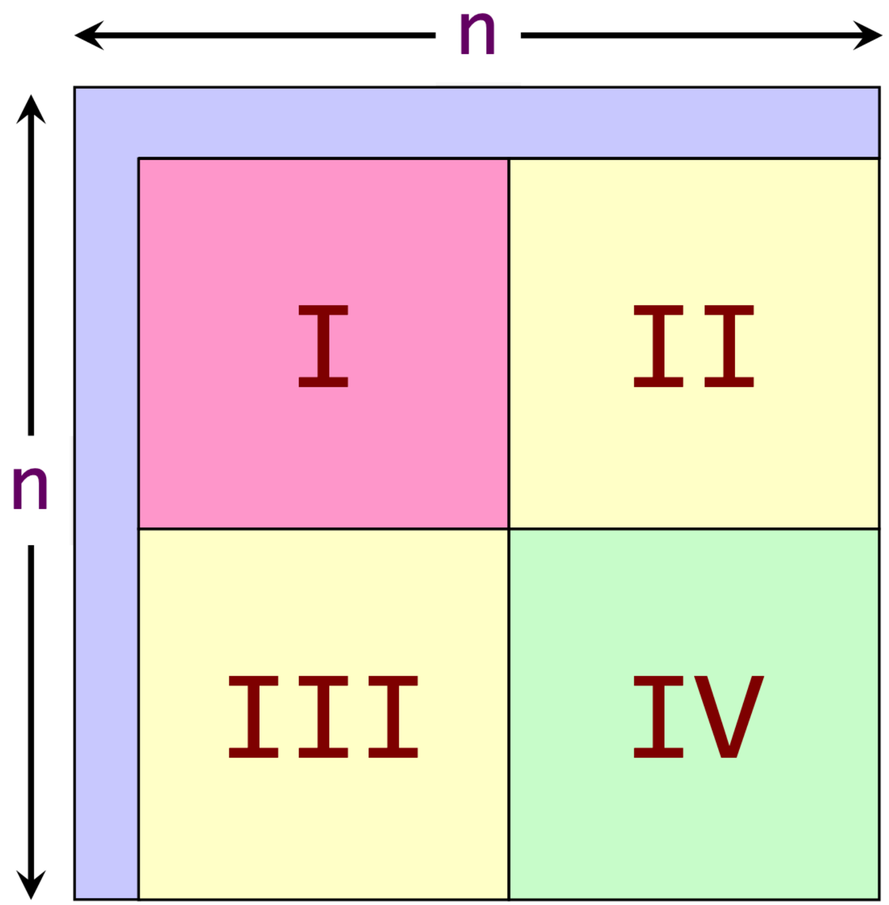
Recursive Construction




```
I;
parallel_spawn II;
III;
parallel_sync;
IV;
```

What is the recurrence for the span?

Recursive Construction




```
I;
parallel_spawn II;
III;
parallel_sync;
IV;
```

Span: $T_\infty(n) = 3T_\infty(n/2) + \Theta(1) = \Theta(n^{\lg 3})$

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_2 3} = n^{\lg 3}$$

$$f(n) = \Theta(1)$$

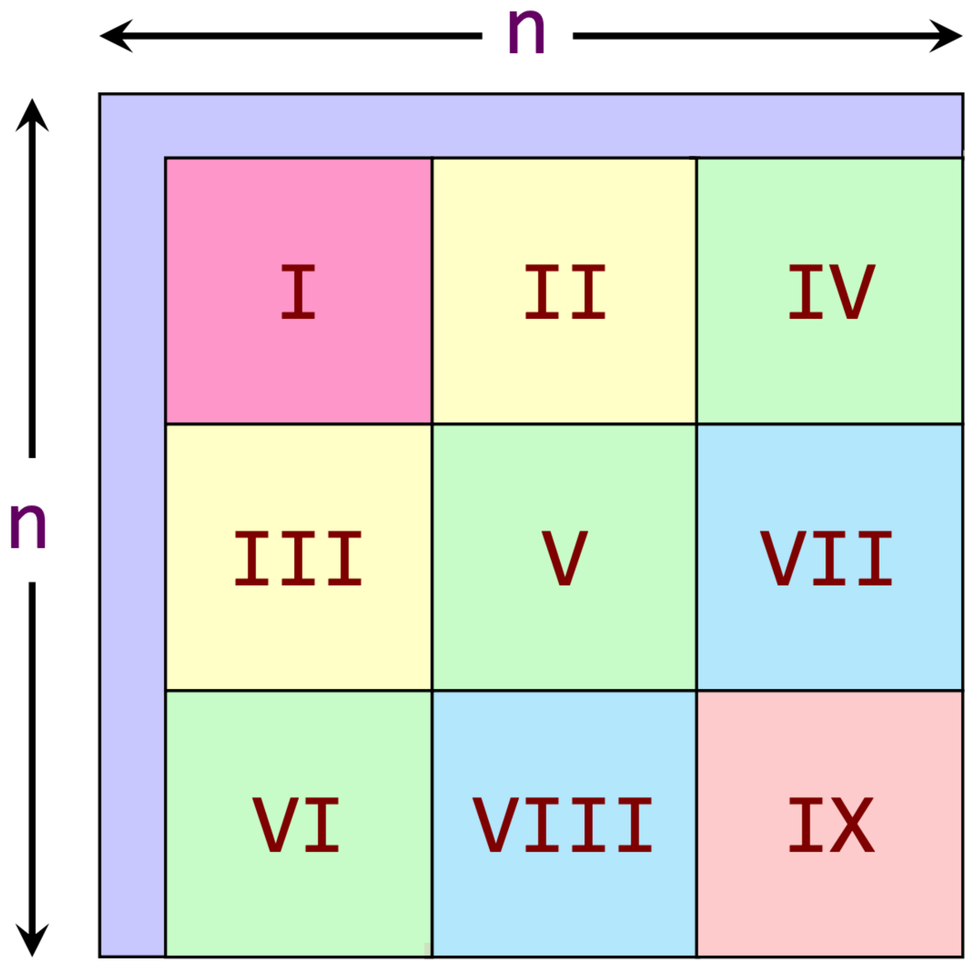
Analysis of Tableau Construction

Work: $T_1(n) = \Theta(n^2)$

Span: $T_\infty(n) = \Theta(\lg^3 n) = O(n^{1.59})$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n^{2-\lg 3}) = \Omega(n^{0.41})$

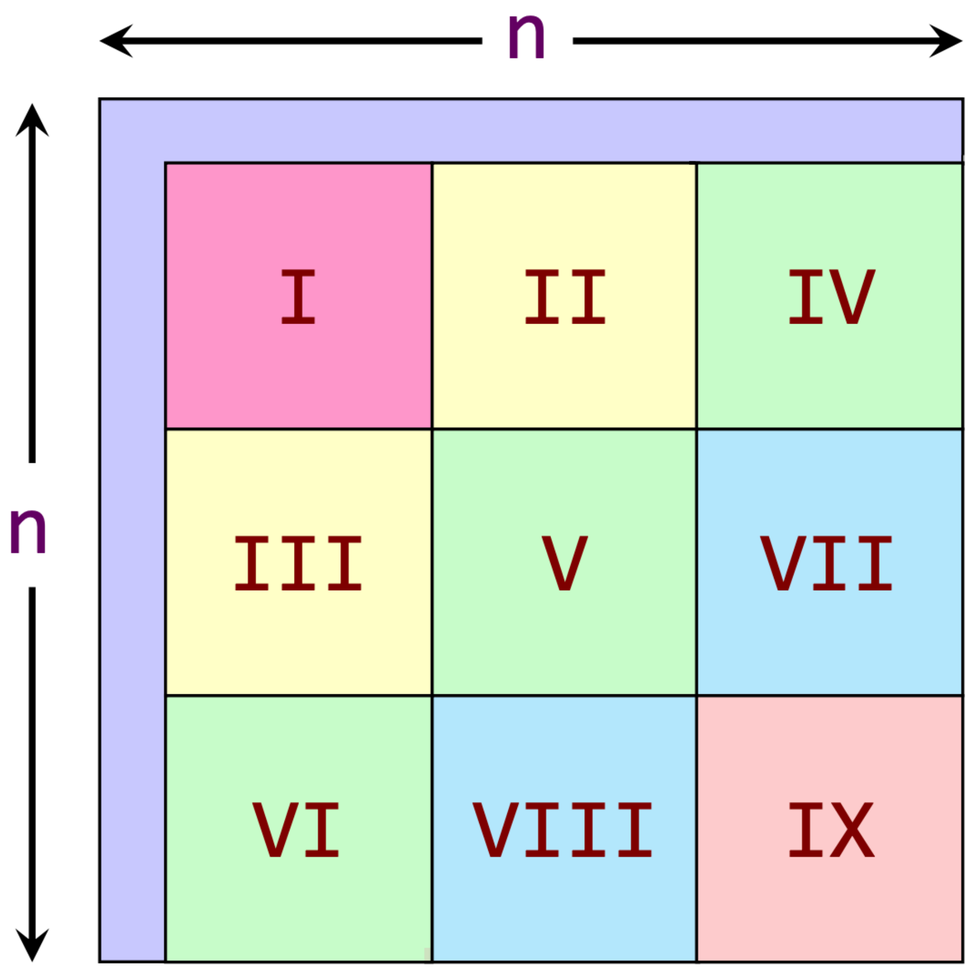
A More Parallel Construction




```
I;
parallel_spawn II;
III;
parallel_sync;
parallel_spawn IV;
parallel_spawn V;
VI;
parallel_sync;
parallel_spawn VII;
VIII;
parallel_sync;
IX;
```

What is the recurrence for the work?

A More Parallel Construction




```
I;
parallel_spawn II;
III;
parallel_sync;
parallel_spawn IV;
parallel_spawn V;
VI;
parallel_sync;
parallel_spawn VII;
VIII;
parallel_sync;
IX;
```

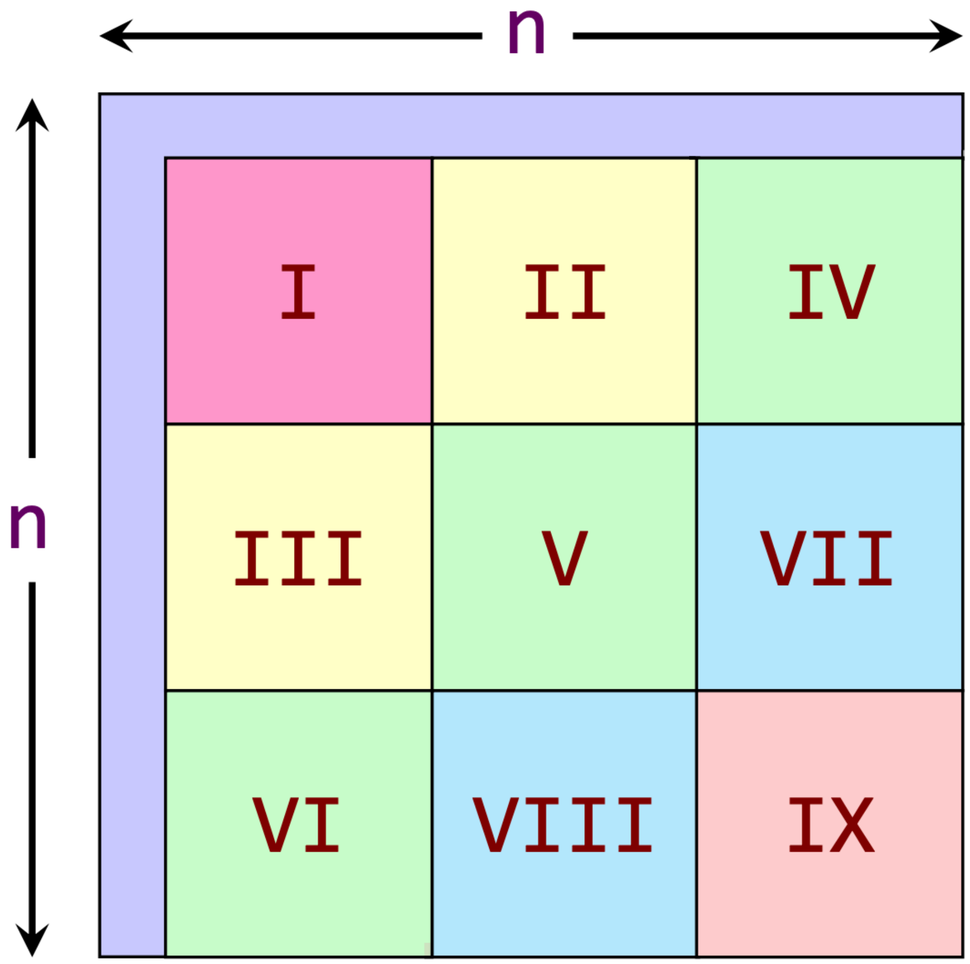
Work: $T_1(n) = 9T_1(n/3) + \Theta(1) = \Theta(n^2)$

Case 1 of Master Method

$n^{\log_b a} = n^{\log_3 9} = n^2$

$f(n) = \Theta(1)$

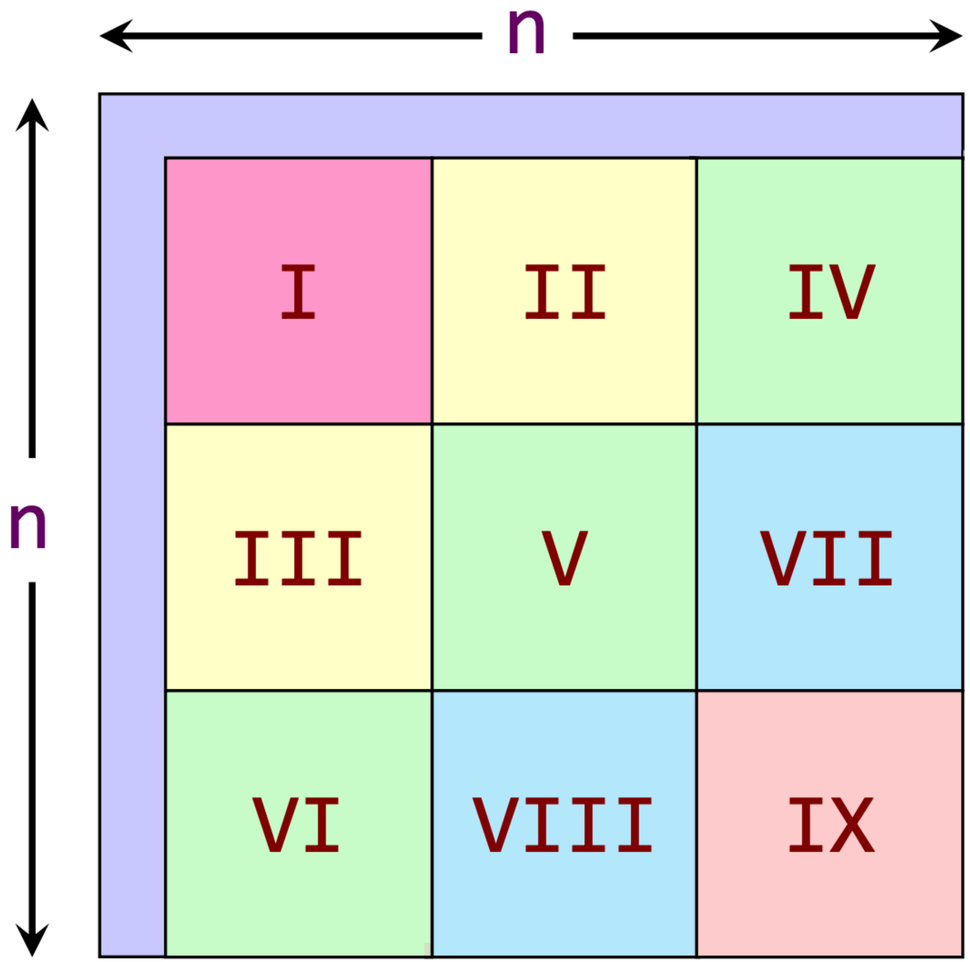
A More Parallel Construction




```
I;
parallel_spawn II;
III;
parallel_sync;
parallel_spawn IV;
parallel_spawn V;
VI;
parallel_sync;
parallel_spawn VII;
VIII;
parallel_sync;
IX;
```

What is the recurrence for the span?

A More Parallel Construction




```
I;
parallel_spawn II;
III;
parallel_sync;
parallel_spawn IV;
parallel_spawn V;
VI;
parallel_sync;
parallel_spawn VII;
VIII;
parallel_sync;
IX;
```

Work: $T_\infty(n) = 5T_\infty(n/3) + \Theta(1) = \Theta(n^{\log_3 5})$

Case 1 of Master Method

$n^{\log_b a} = n^{\log_3 5} = n^{\log_3 5}$

$f(n) = \Theta(1)$

Analysis of Revised Method

Work: $T_1(n) = \Theta(n^2)$

Span: $T_\infty(n) = \Theta(\log_3 5) = O(n^{1.47})$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n^{2-\log_3 5}) = \Omega(n^{0.53})$

Nine-way divide-and-conquer has about $\Omega(n^{0.12})$ more parallelism than four-way divide-and-conquer, but it exhibits less cache locality

Puzzle

What is the largest parallelism that can be obtained for the tableau-construction problem in the binary forking model?

- You may only use basic fork-join control constructs (e.g., `parallel_spawn, parallel_sync, parallel_for`) for synchronization.

- No using locks, atomic instructions, synchronizing through memory, etc.

CSE 6220/CX 4220
Introduction to HPC

Lecture 4: Caching and Cache-Efficient Algorithms

Helen Xu
hxu615@gatech.edu

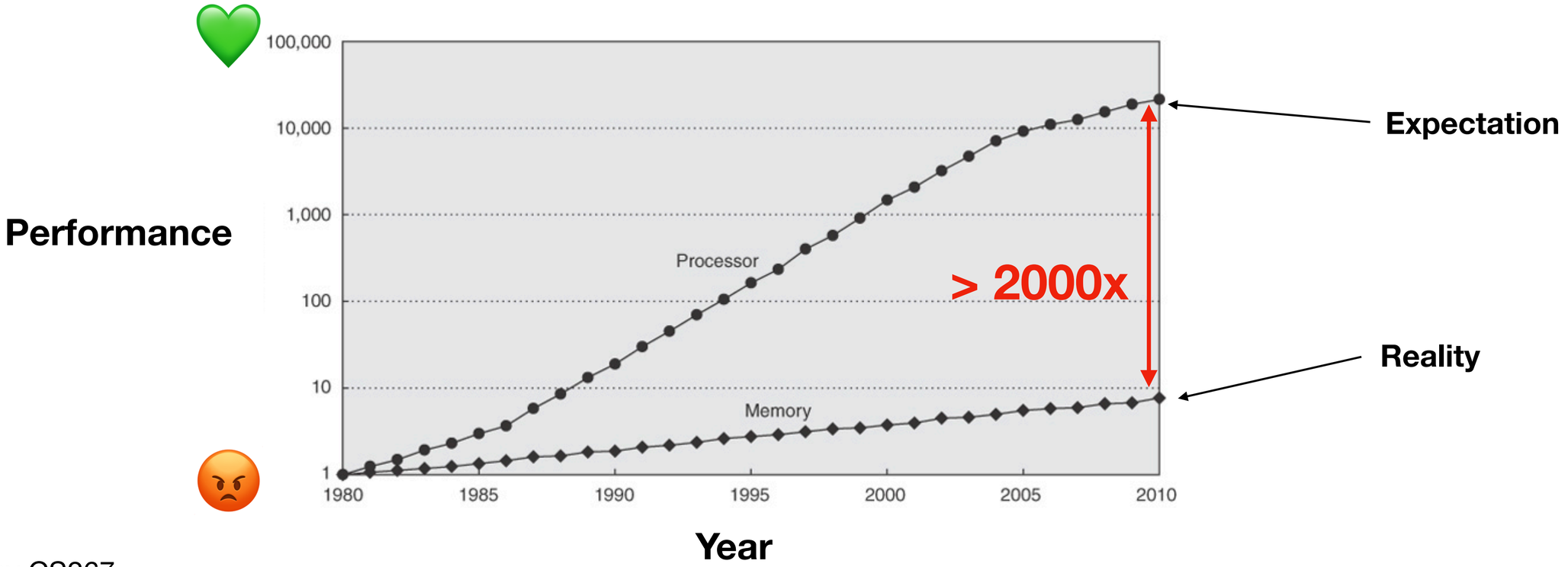
Georgia Tech College of Computing
School of Computational Science and Engineering

Let’s discuss the memory hierarchy on one processor - why?

Memory limitations affect overall performance

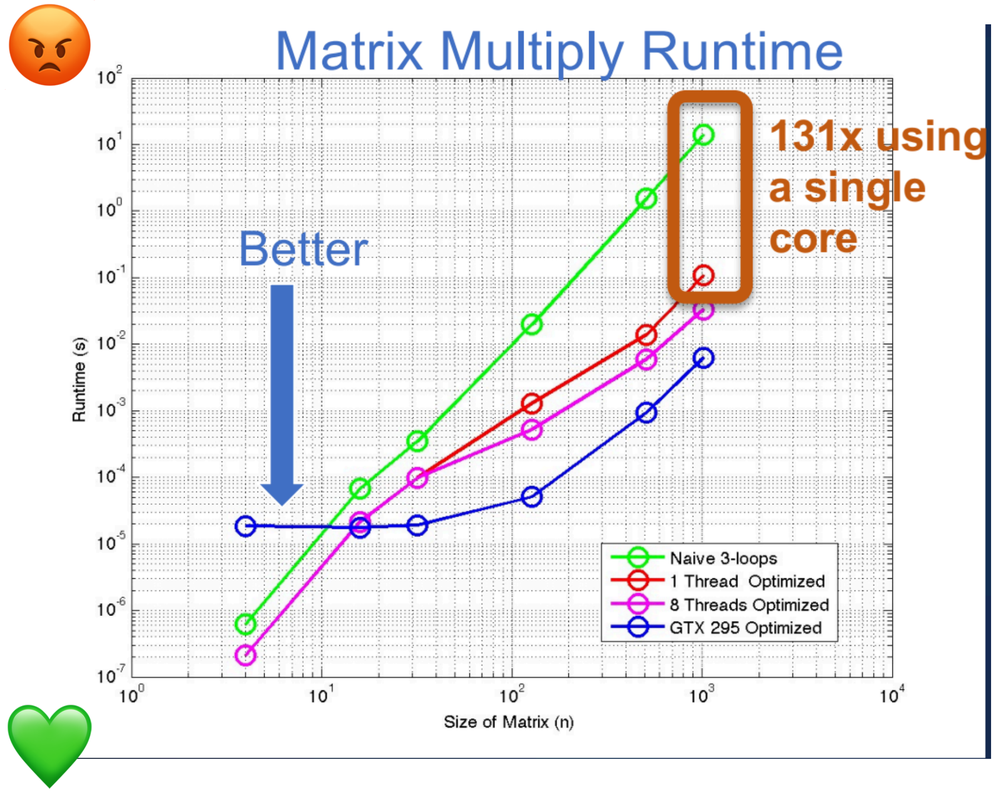
Most applications run at **<10%** of “peak” performance (max possible rate of flops).

Much of the performance is lost on a single processor due to **data movement**.



From UC Berkeley CS267

Why should we care about serial performance in a course on high-performance computing?



Starting with a fast serial implementation is an **important first step** towards a fast parallel implementation.

Another way to say it: **If you parallelize slow serial code, you will get slow parallel code.**

Matrix multiply is extreme, but ~10x improvement from **sequential optimizations** is more common.

Memory Hierarchies

Memory access performance measurements

Memory accesses (load/store) have two costs:

Latency - the cost to load or store one word (α)

Bandwidth - the average rate (bytes / sec) to load/store a large chunk of data (β)

Bandwidth

≈ data throughput (bytes/second)

Low Bandwidth

High Bandwidth

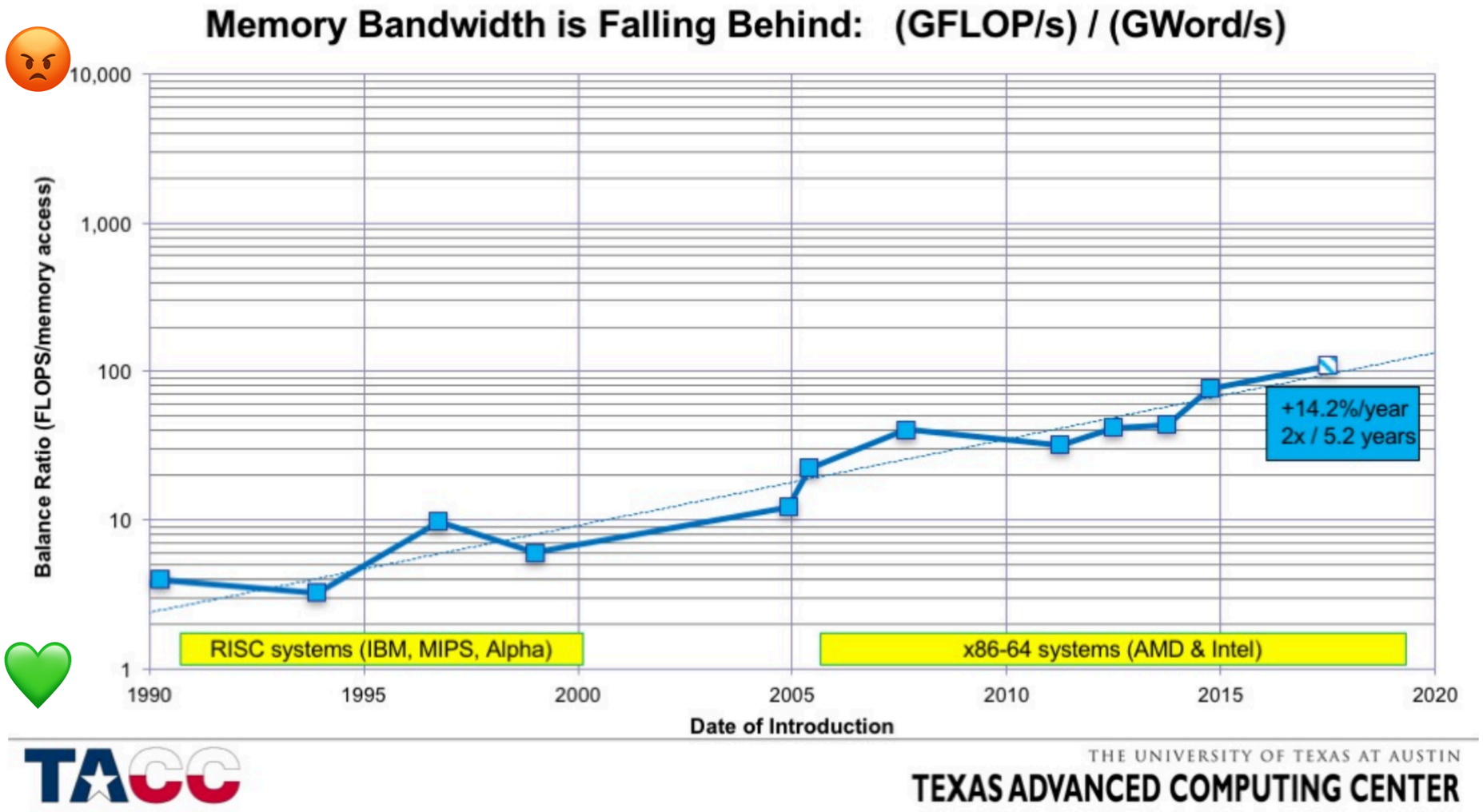
Latency

≈ delay due data travel time (secs)

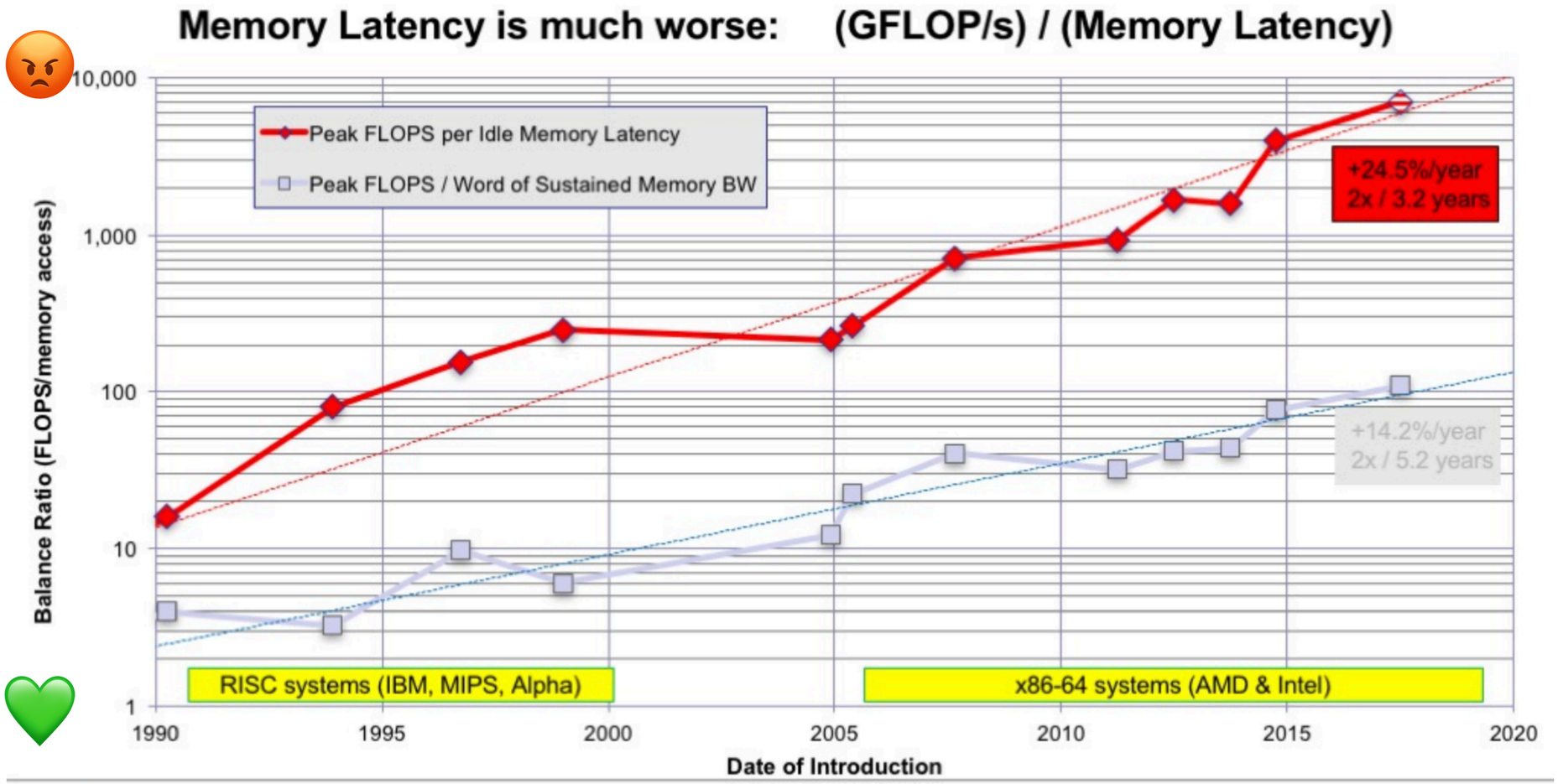
Low Latency

High Latency

Memory bandwidth gap



Memory latency gap is worse



TACC

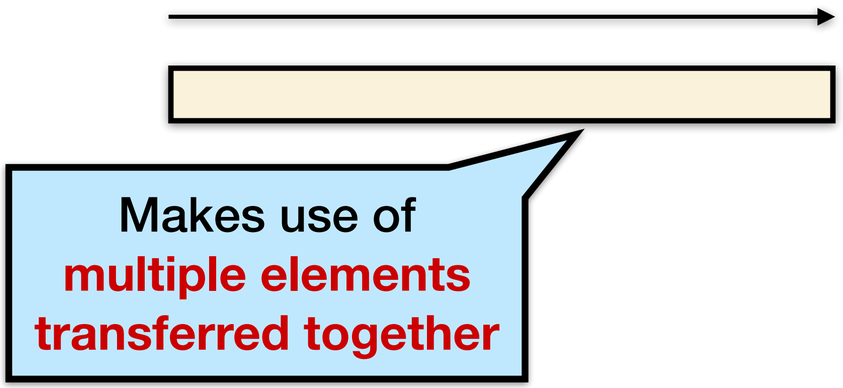
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

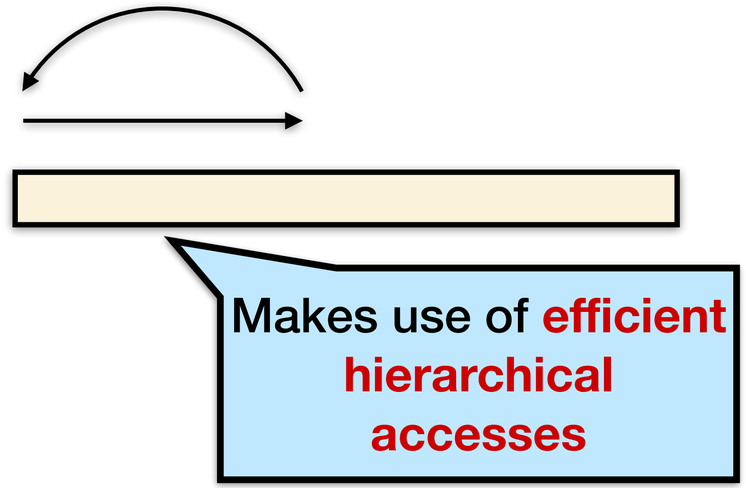
Two main types of locality: Spatial and Temporal

Most programs have a high degree of **locality**.

Spatial locality: how many accesses an algorithm makes to **nearby** data over a short period of time [Denning72, Denning05].

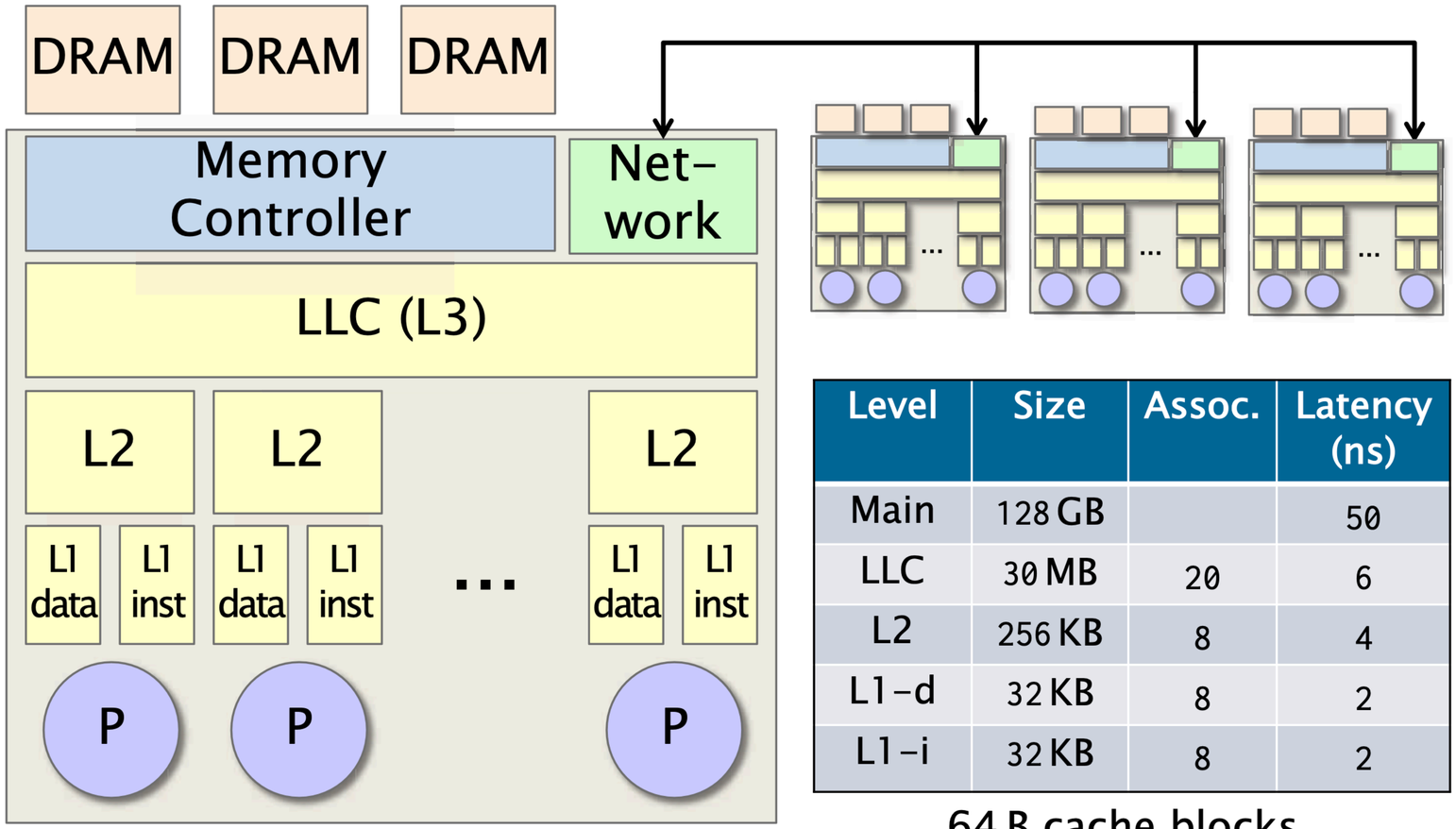
Temporal locality: how many repeated accesses an algorithm makes to **the same** data over a short period of time [Denning72, Denning05].





Memory hierarchy

The **memory hierarchy** takes advantage of locality to speed up the average case to handle memory latency.



Level	Size	Assoc.	Latency (ns)
Main	128 GB		50
LLC	30 MB	20	6
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64 B cache blocks

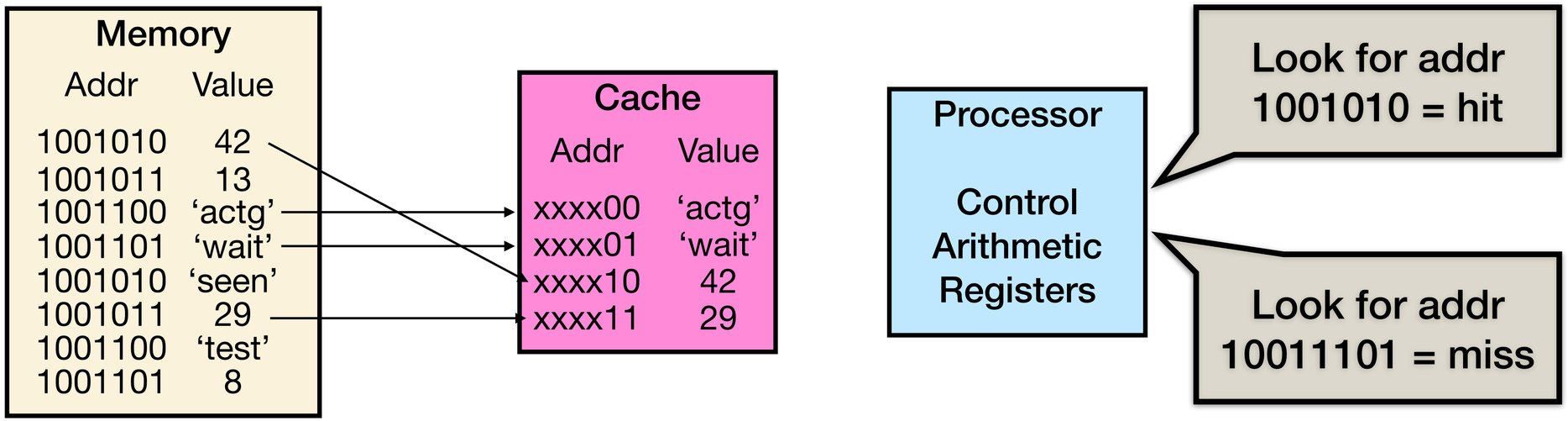
Figure from MIT 6.172

Cache basics

Cache is fast (expensive) memory which keeps a copy of the data; it is hidden from software.

Cache-line length: number of bytes loaded together in one entry (often 64 bytes).

Simple example: data at address xxxxx10 is stored at cache location 10.

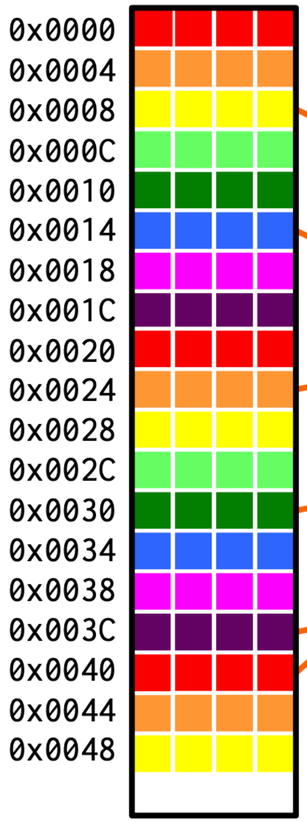


Cache **hit**: access to a memory address in cache - cheap

Cache **miss**: non-cached memory access - expensive

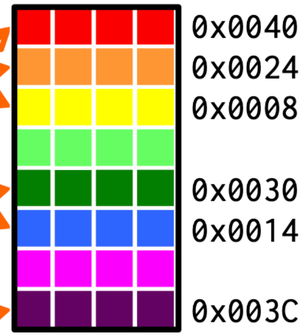
Need to look in next, slower level of memory.

Direct-mapped cache



w–bit address space

Cache size $\mathcal{M} = 32$.
Line/block size $\mathcal{B} = 4$.



tag

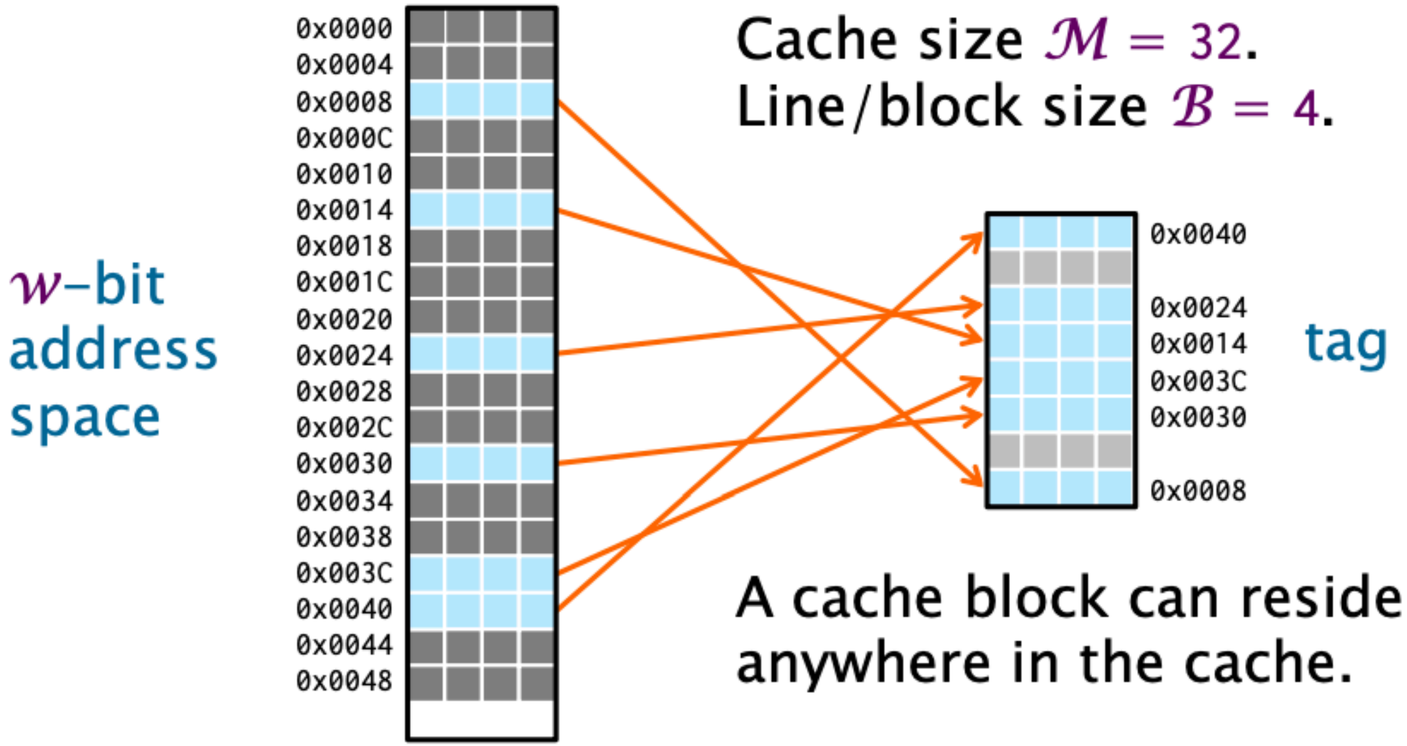
A cache block's set determines its location in the cache.

address

	tag	set	offset
bits	$w - \lg \mathcal{M}$	$\lg(\mathcal{M}/\mathcal{B})$	$\lg \mathcal{B}$

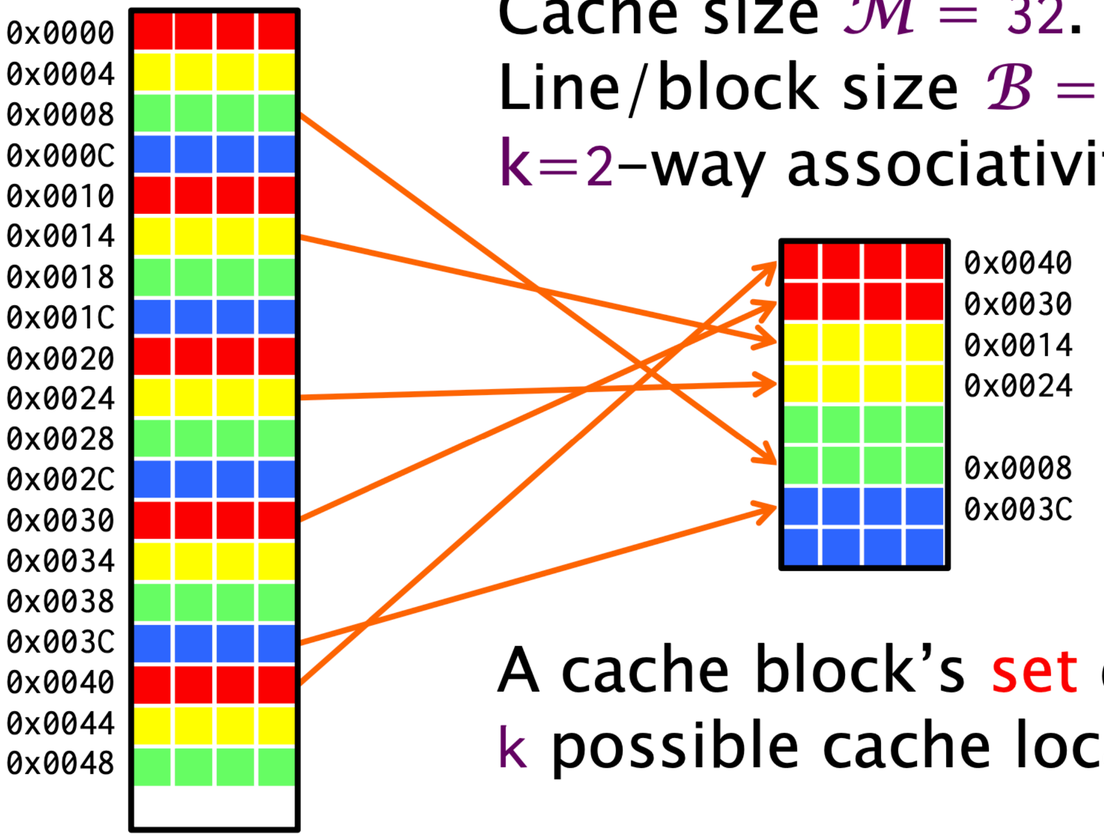
To find a block in the cache, only a single location in the cache need be searched.

Fully-associative cache



To find a block in the cache, the entire cache must be searched for the tag. When the cache becomes full, a block must be **evicted** to make room for a new block. The **replacement policy** determines which block to evict.

Set-associative cache



w-bit address space

Cache size $\mathcal{M} = 32$.
Line/block size $\mathcal{B} = 4$.
k=2-way associativity.

tag

A cache block's set determines k possible cache locations.

address

	tag	set	offset
bits	w – lg($\mathcal{M}$/k)	lg($\mathcal{M}$/k$\mathcal{B}$)	lg $\mathcal{B}$

To find a block in the cache, only the k locations of its set must be searched.

Type of cache misses - Three C's

Cold miss

The first time a cache block is accessed.

Capacity miss

The previous cached copy would have been evicted even with a fully-associative cache.

Conflict miss

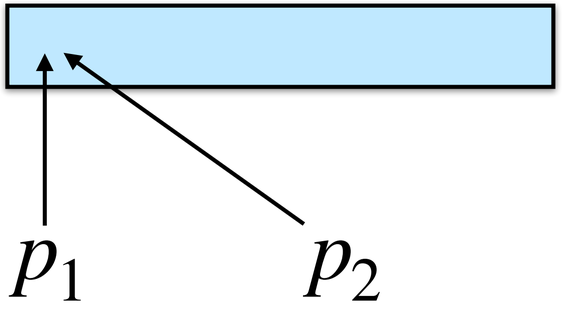
Too many blocks from the same set in cache. The block would not have been evicted with a fully-associative cache.

Type of cache misses - Three C's

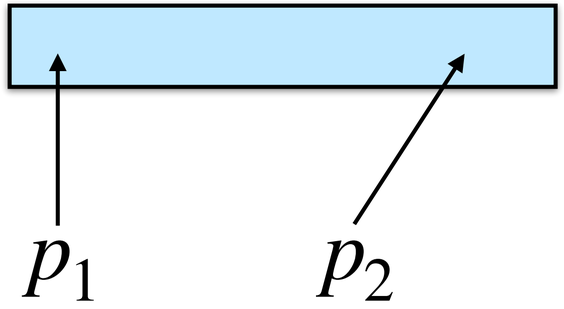
Sharing miss

- Another processor acquired exclusive access to the cache block.
- True-sharing miss: The two processors are accessing the same data on the cache line.
- False-sharing miss: The two processors are accessing different data that happen to reside on the same cache line.

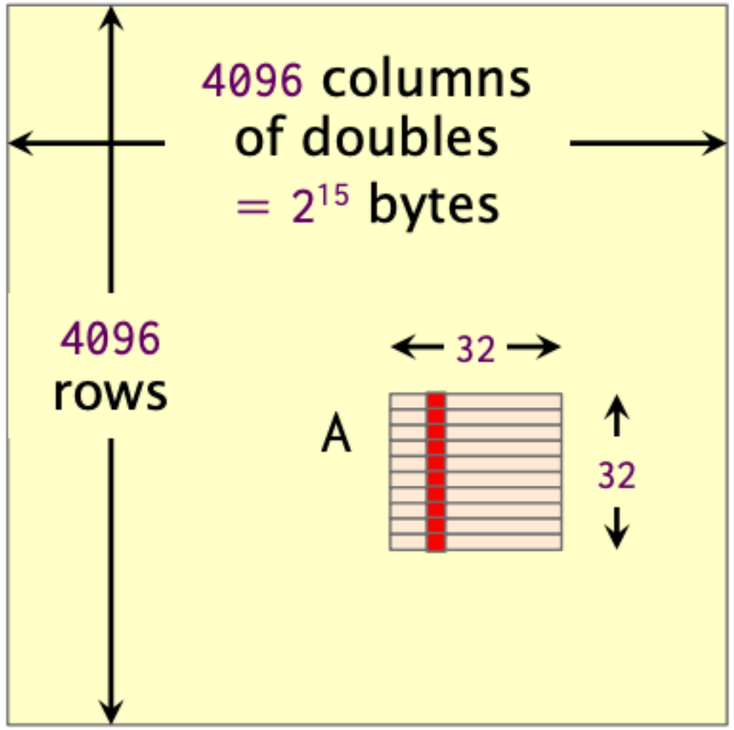
True Sharing



False Sharing



Conflict misses for submatrices



Assume:

- Word width $w = 64$.
- Cache size $\mathcal{M} = 32\text{K}$.
- Line (block) size $\mathcal{B} = 64$.
- k=4–way associativity.

Conflict misses can be problematic for caches with limited associativity.

address

	tag	set	offset
bits	$w - \lg(\mathcal{M}/k)$	$\lg(\mathcal{M}/k\mathcal{B})$	$\lg \mathcal{B}$
	51	7	6

Analysis

Look at a column of submatrix A.
The addresses of the elements are
x, $x+2^{15}$, $x+2\cdot2^{15}$, …, $x+31\cdot2^{15}$.
They all fall into the same set!

Solutions

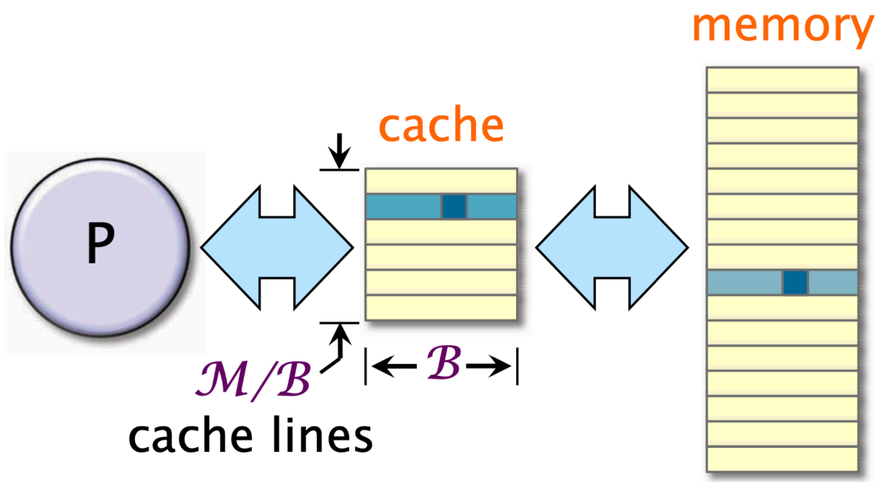
Copy A into a temporary 32×32 matrix, or pad rows.

Ideal-Cache Model

Ideal-Cache Model

Parameters

- Two-level hierarchy
- Cache size of $\mathscr{M}$ bytes
- Cache-line length of $\mathscr{B}$ bytes
- Fully associative
- Optimal, omniscient replacement.



Performance Measures

- **Work** W (ordinary running time)
- **Cache misses** Q (number of cache lines that need to be transferred between cache and memory)

How reasonable are ideal caches?

"LRU" Lemma [ST85]. Suppose that an algorithm incurs Q cache misses on an ideal cache of size $\mathcal{M}$. Then on a fully associative cache of size $2\mathcal{M}$ that uses the **least-recently used (LRU)** replacement policy, it incurs at most $2Q$ cache misses.

Implication

For asymptotic analyses, one can assume optimal or LRU replacement, as convenient.

Software engineering

- Design a **theoretically good** algorithm.
- **Engineer** for detailed performance.
  - Real caches are not fully associative.
  - Loads and stores have different costs with respect to bandwidth and latency.

Cache-miss lemma

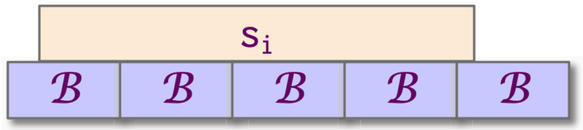
Lemma. Suppose that a program reads a set of r data segments, where the ith segment consists of s_i bytes, and suppose that

$$\sum_{i=1}^{r} s_i = N < \mathcal{M}/3 \text{ and } N/r \geq \mathcal{B} .$$

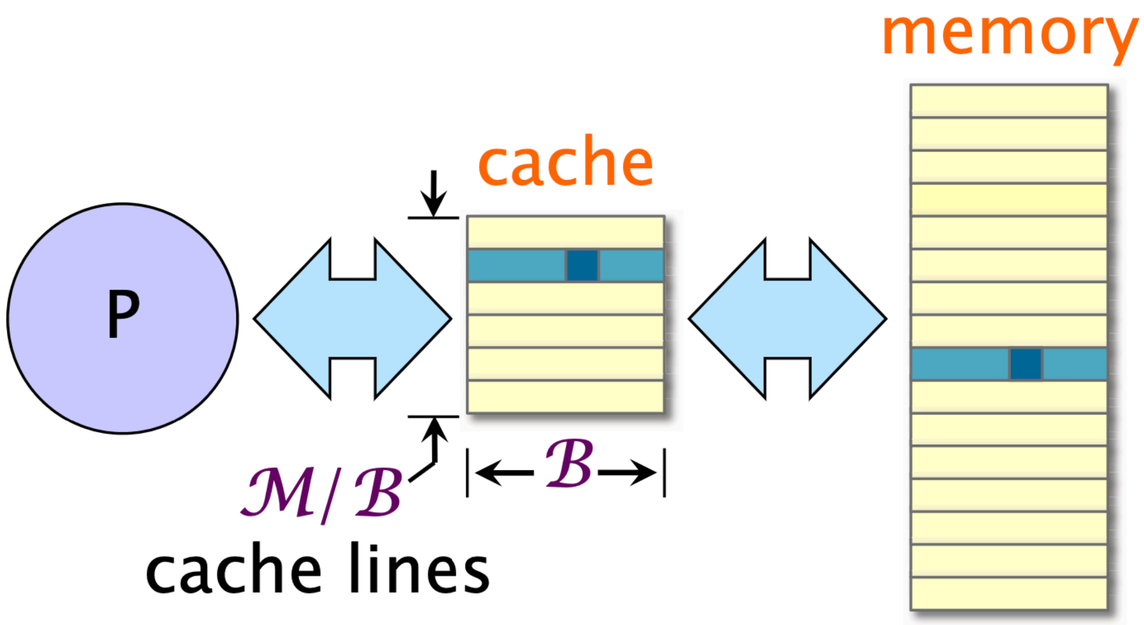
Then all of the segments fit into cache, and the number of misses to read them all is at most $3N/\mathcal{B}$.

Proof. Suppose that a program reads a set of r data segments, where the ith segment consists of A single segment s_i incurs at most $s_i/\mathcal{B}$ misses, and so

$$\begin{aligned}
\sum_{i=1} (s_i/\mathcal{B} + 2) &= N/\mathcal{B} + 2r \\
&= (N/\mathcal{B} + 2r\mathcal{B})/\mathcal{B} \\
&\leq N/\mathcal{B} + 2N/\mathcal{B} \\
&= 3N/\mathcal{B} . \quad \blacksquare
\end{aligned}$$



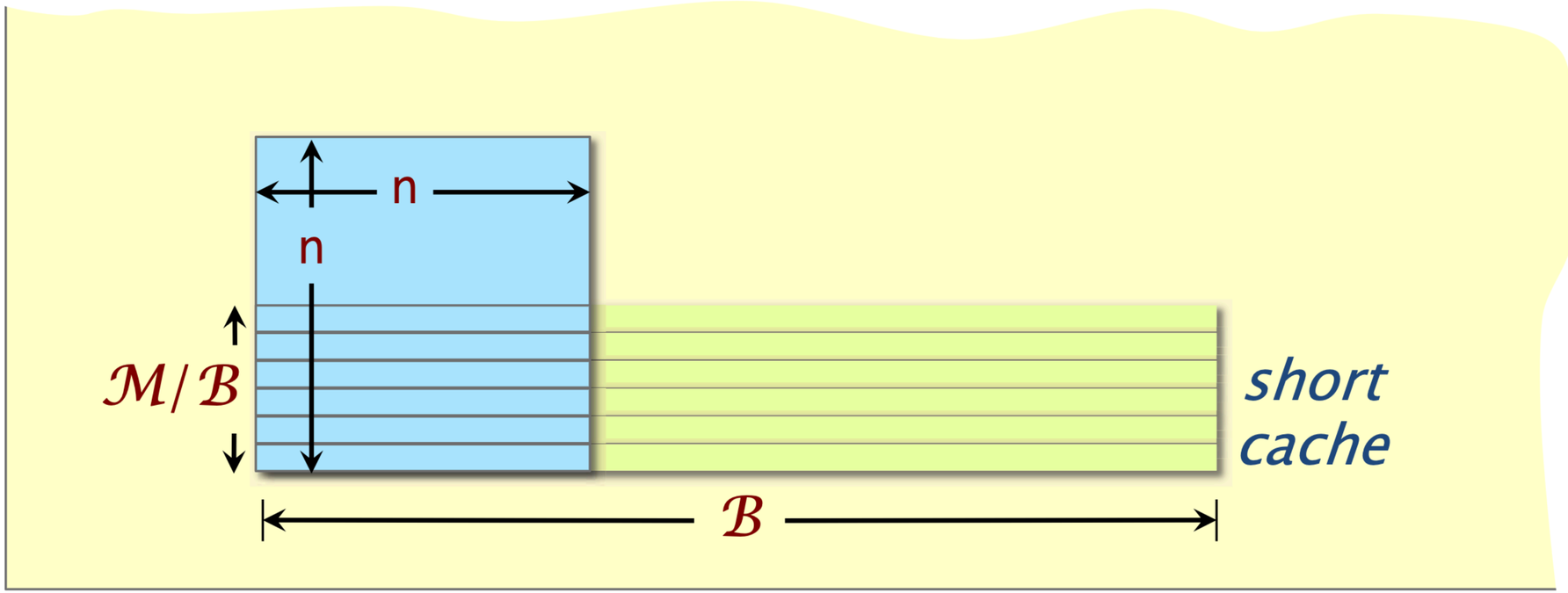
Tall caches



Tall-cache assumption
$\mathcal{B}^2 < c\mathcal{M}$ for some sufficiently small constant $c \leq 1$.

Example: Intel Xeon E5-2666 v3
- Cache-line length = 64 bytes.
- L1-cache size = 32 Kbytes.

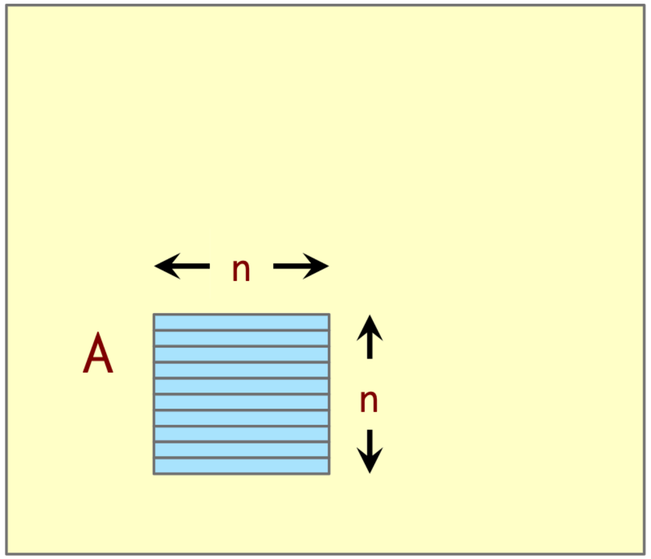
What's wrong with short caches?



Tall-cache assumption
$\mathcal{B}^2 < c\mathcal{M}$ for some sufficiently small constant $c \leq 1$.

An $n \times n$ submatrix stored in row-major order may not fit in a short cache even if $n^2 < c\mathcal{M}$!

Submatrix caching lemma



Lemma. Suppose that an $n \times n$ submatrix A is read into a tall cache satisfying $\mathcal{B}^2 < c\mathcal{M}$, where $c \leq 1$ is constant, and suppose that $c\mathcal{M} \leq n^2 < \mathcal{M}/3$. Then A fits into cache, and the number of misses to read all A's elements is at most $3n^2/\mathcal{B}$.

Proof. We have $N = n^2$, $n = r = s_i$, $\mathcal{B} \leq n = N/r$, and $N < \mathcal{M}/3$. Thus, the Cache-Miss Lemma applies. ■

Cache Analysis of Matrix Multiplication

Why matrix multiplication?

Matrix multiplication is an **important kernel** in many problems:

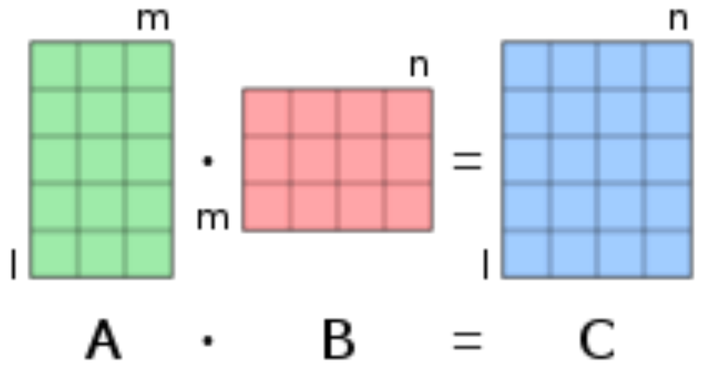
– Dense linear algebra is a motif in every list,

– Closely related to other algorithms, e.g., transitive closure on a graph,

– And dominates training time in deep learning (CNNs)

Good model problem (well-studied, illustrates ideas).

Easy to find good libraries that are hard to beat! (e.g., Intel MKL, etc.)



Multiply square matrices

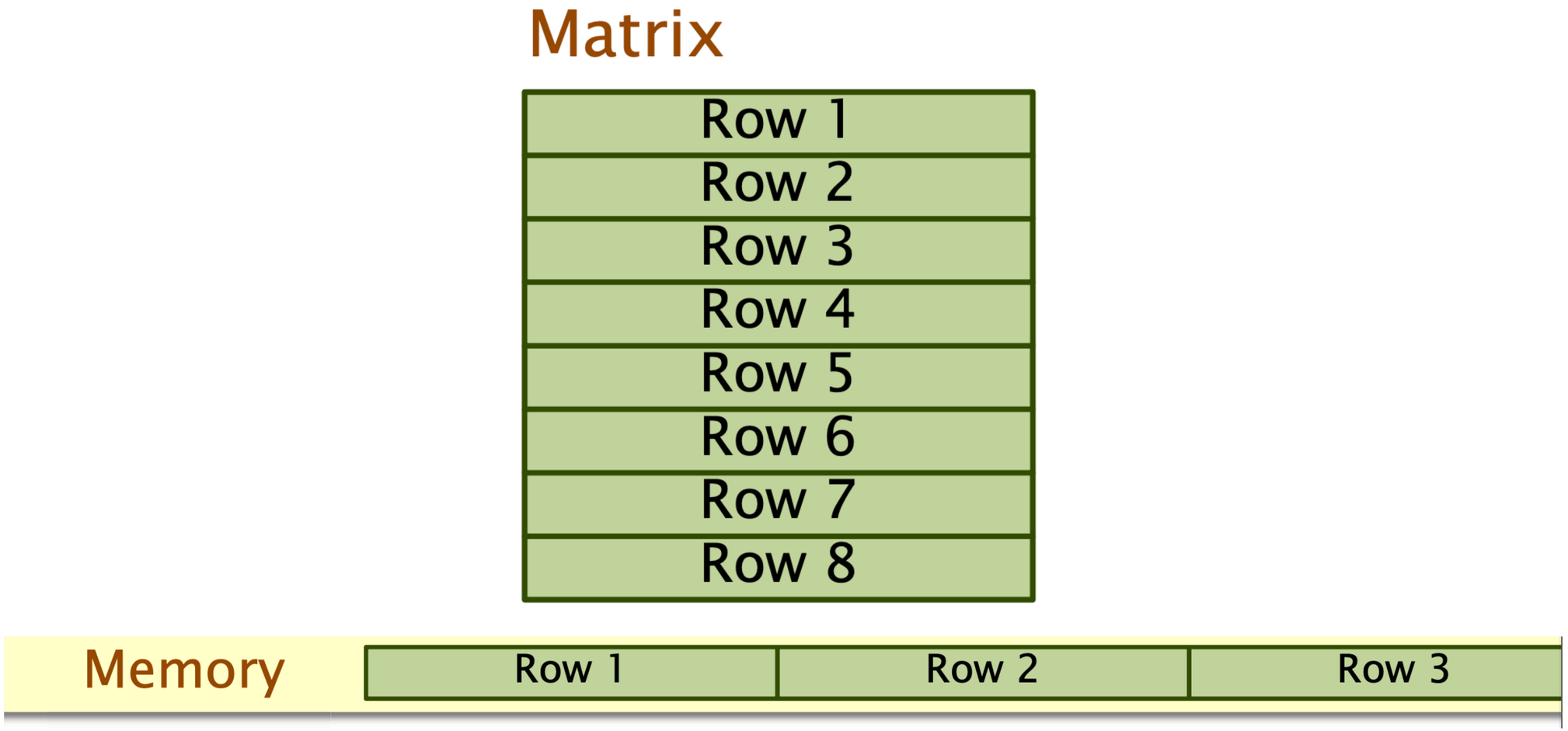
```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Analysis of work:

$W(n) = \Theta(n^3)$

Memory layout of matrices

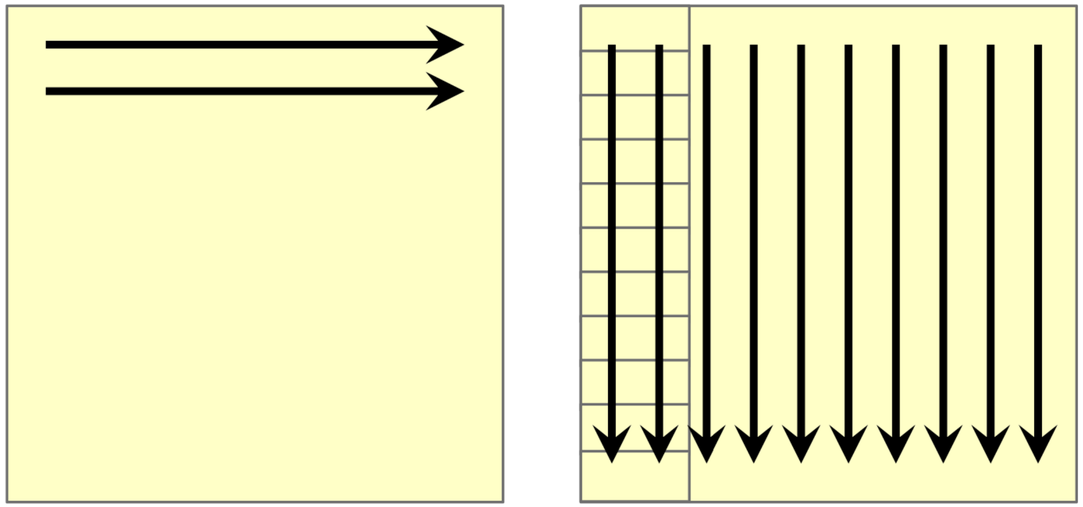
In this matrix-multiplication code, matrices are laid out in memory in **row-major order**.



Analysis of cache misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache



A B

Case 1

$n > c\mathscr{M}/\mathscr{B}$. Analyze matrix B.

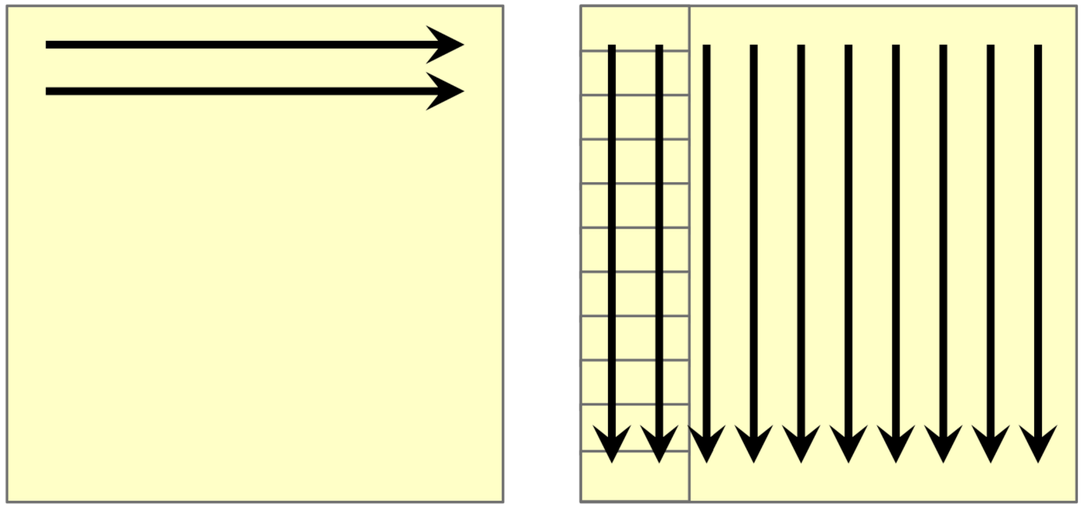
Assume LRU.

$Q(n) = \Theta(n^3)$, since matrix B misses on every access.

Analysis of cache misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache



A B

Case 2

$c'\mathscr{M}^{1/2} < n < c\mathscr{M}/\mathscr{B}$. Analyze matrix B.

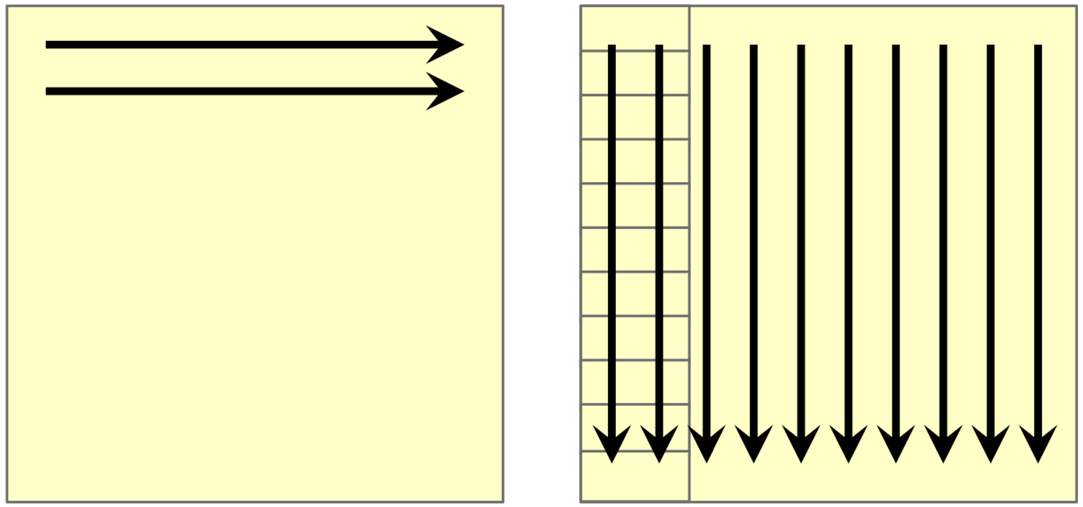
Assume LRU.

$Q(n) = n \cdot \Theta(n^2/\mathscr{B}) = \Theta(n^3/\mathscr{B})$, since matrix B can exploit spatial locality.

Analysis of cache misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache



A B

Case 3

$n < c'\mathscr{M}^{1/2}$. Analyze matrix B.

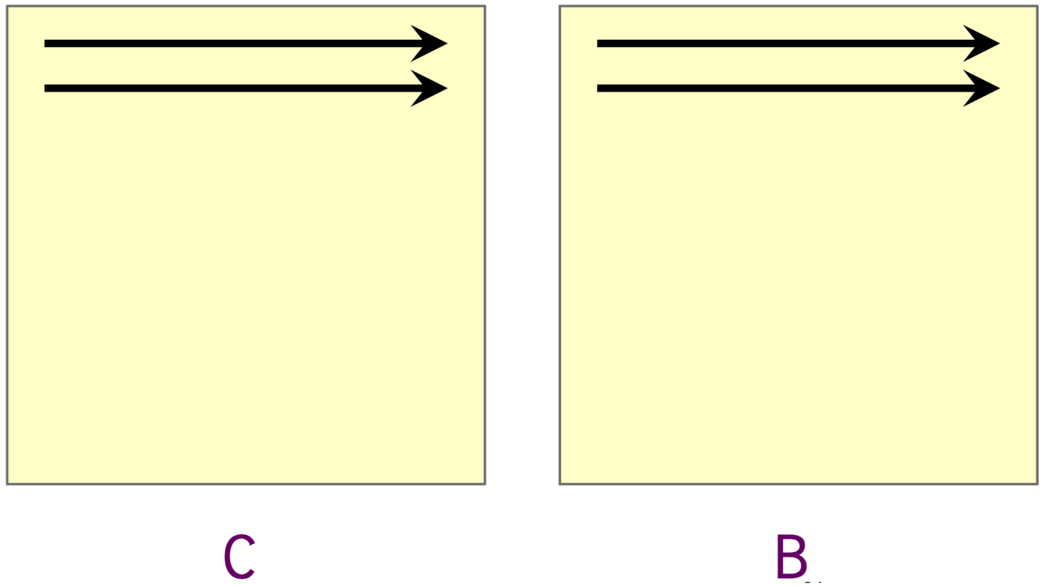
Assume LRU.

$Q(n) = \Theta(n^2/\mathscr{B})$, since everything fits in cache!

Swapping inner loop order

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; I++)
    for (int64_t k=0; k < n; k++)
      for (int64_t j=0; j < n; j++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache



Assume matrix B. Assume LRU.

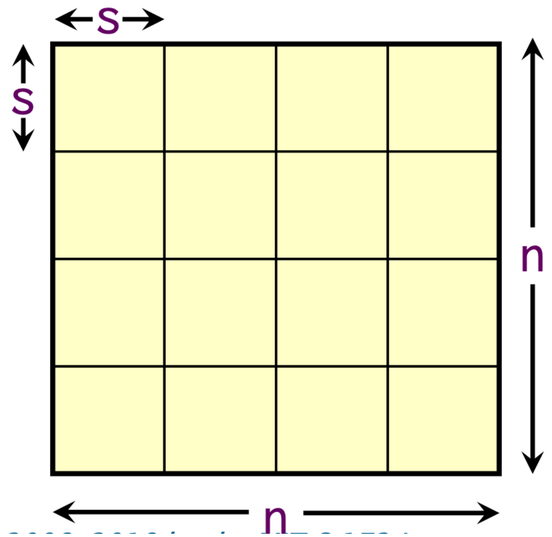
$$Q(n) = n \cdot \Theta(n^2/\mathscr{B}) = \Theta(n^3/\mathscr{B}),$$

since matrix B can exploit spatial locality.

Tiling (aka Blocking)

Tiled (Blocked) matrix multiply

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1; i<i1+s && i<n; i++)
          for (int64_t j=j1; j<j1+s && j<n; j++)
            for (int64_t k=k1; k<k1+s && k<n; k++)
              C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```



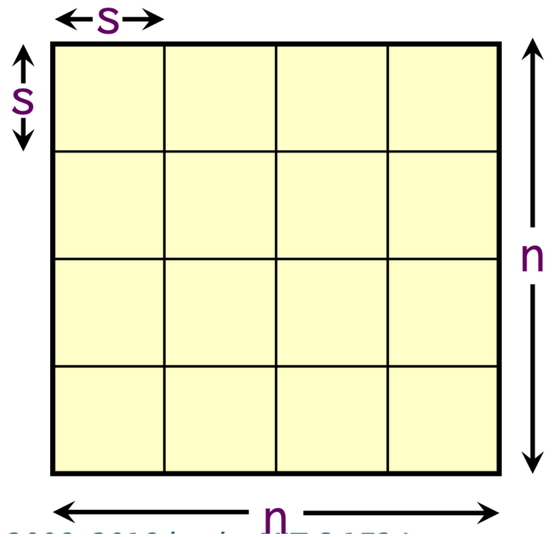
Analysis of work:

$$W(n) = \Theta((n/s)^3(s)^3)$$
$$= \Theta(n^3)$$

Tile size (or block size)

Tiled (Blocked) matrix multiply

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1; i<i1+s && i<n; i++)
          for (int64_t j=j1; j<j1+s && j<n; j++)
            for (int64_t k=k1; k<k1+s && k<n; k++)
              C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```



Analysis of cache misses

Tune s so that the submatrices just fit into cache:

$$s = \Theta(\mathscr{M}^{1/2}).$$

Submatrix Caching Lemma implies $\Theta(s^2/\mathscr{B})$ misses per submatrix.

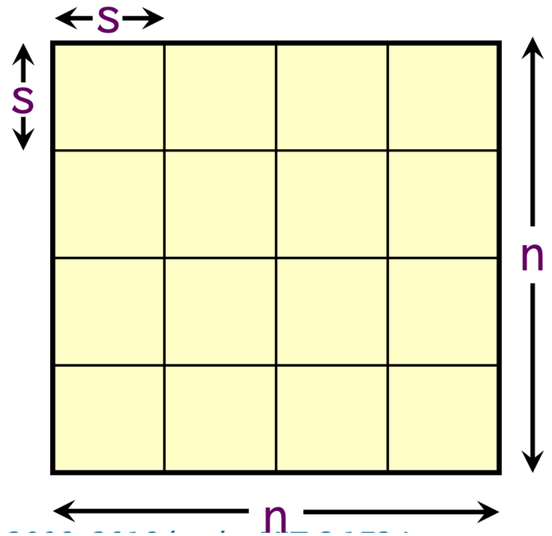
$$Q(n) = \Theta((n/s)^3(s^2/\mathscr{B})) = \Theta(n^3/(\mathscr{B}\mathscr{M}^{1/2})).$$

Optimal [HK81]

Tiled (Blocked) matrix multiply

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1;                    i++)
          for (int64_t j=j                    n; j++)
            for (int64_t k                    k<n; k++)
              C[i*n+j] +=                     n+j];
}
```

How?



Analysis of cache misses

Tune s so that the submatrices just fit into cache:

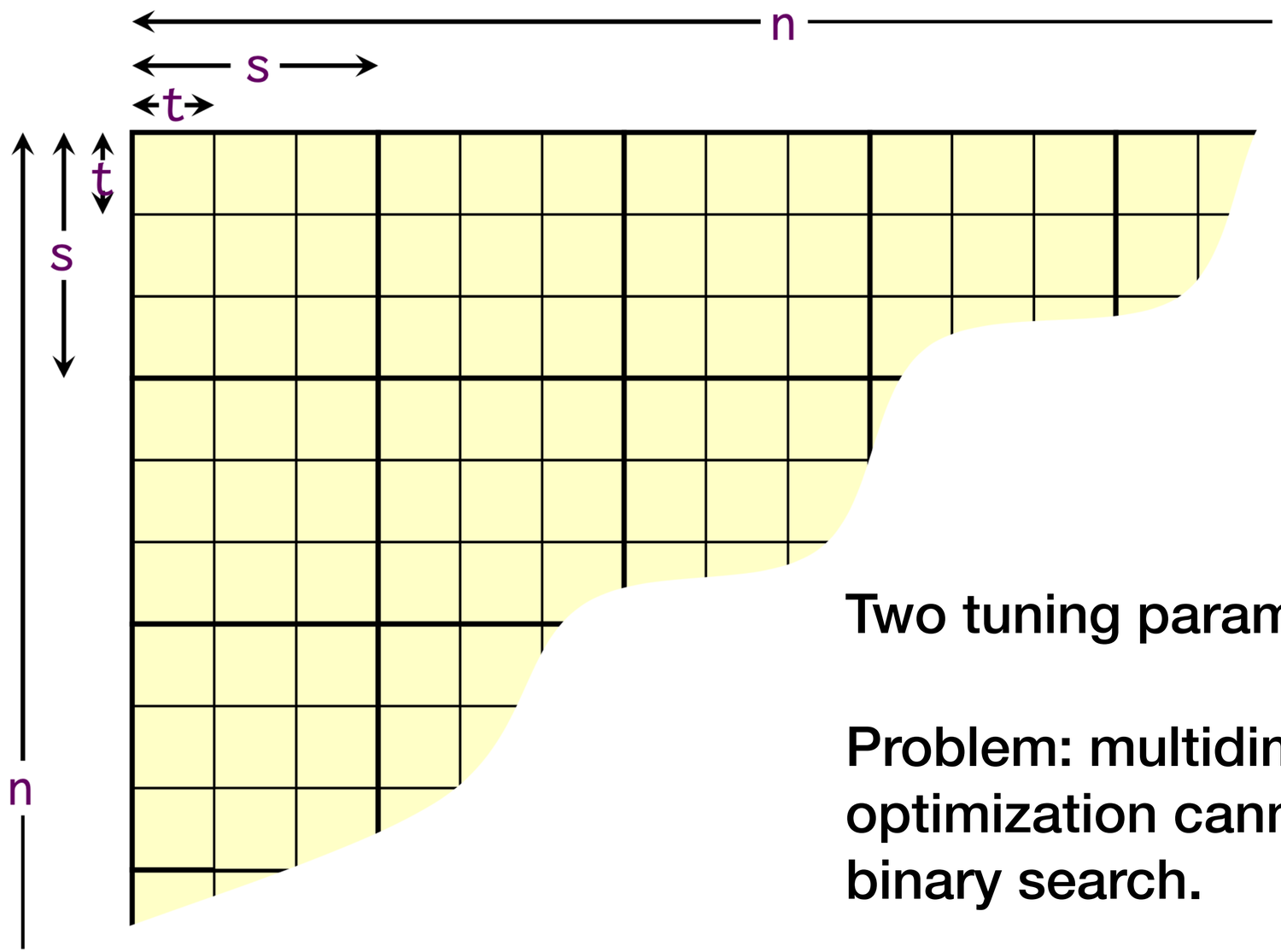
$$s = \Theta(\mathscr{M}^{1/2}).$$

Submatrix Caching Lemma implies $\Theta(s^2/\mathscr{B})$ misses per submatrix.

Optimal [HK81]

$$Q(n) = \Theta((n/s)^3(s^2/\mathscr{B})) = \Theta(n^3/(\mathscr{B}\mathscr{M}^{1/2})).$$

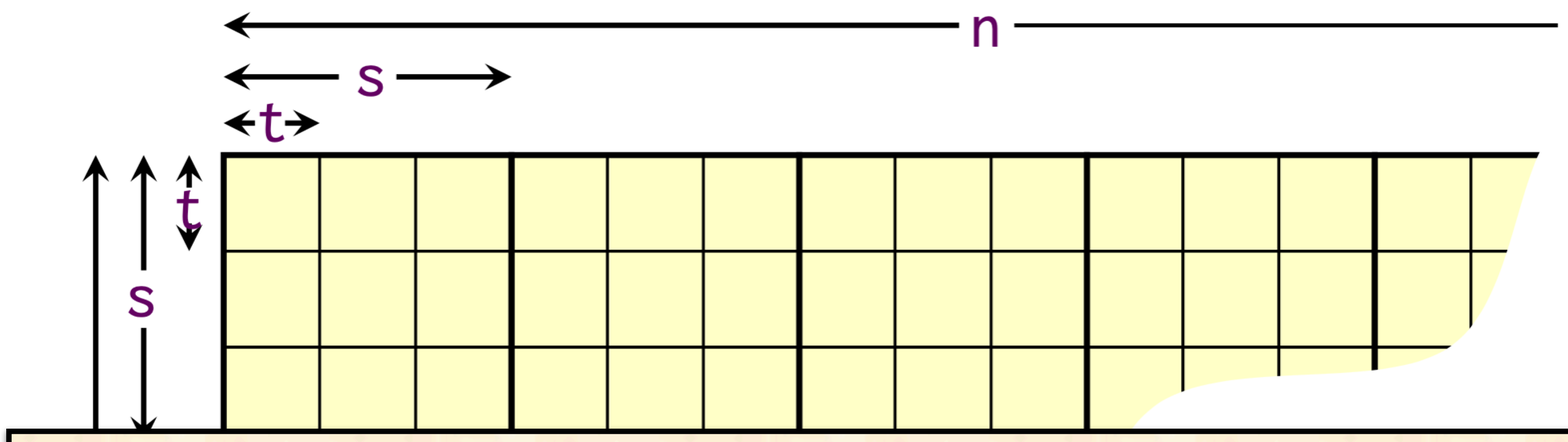
Two-level cache



Two tuning parameters: s and t.

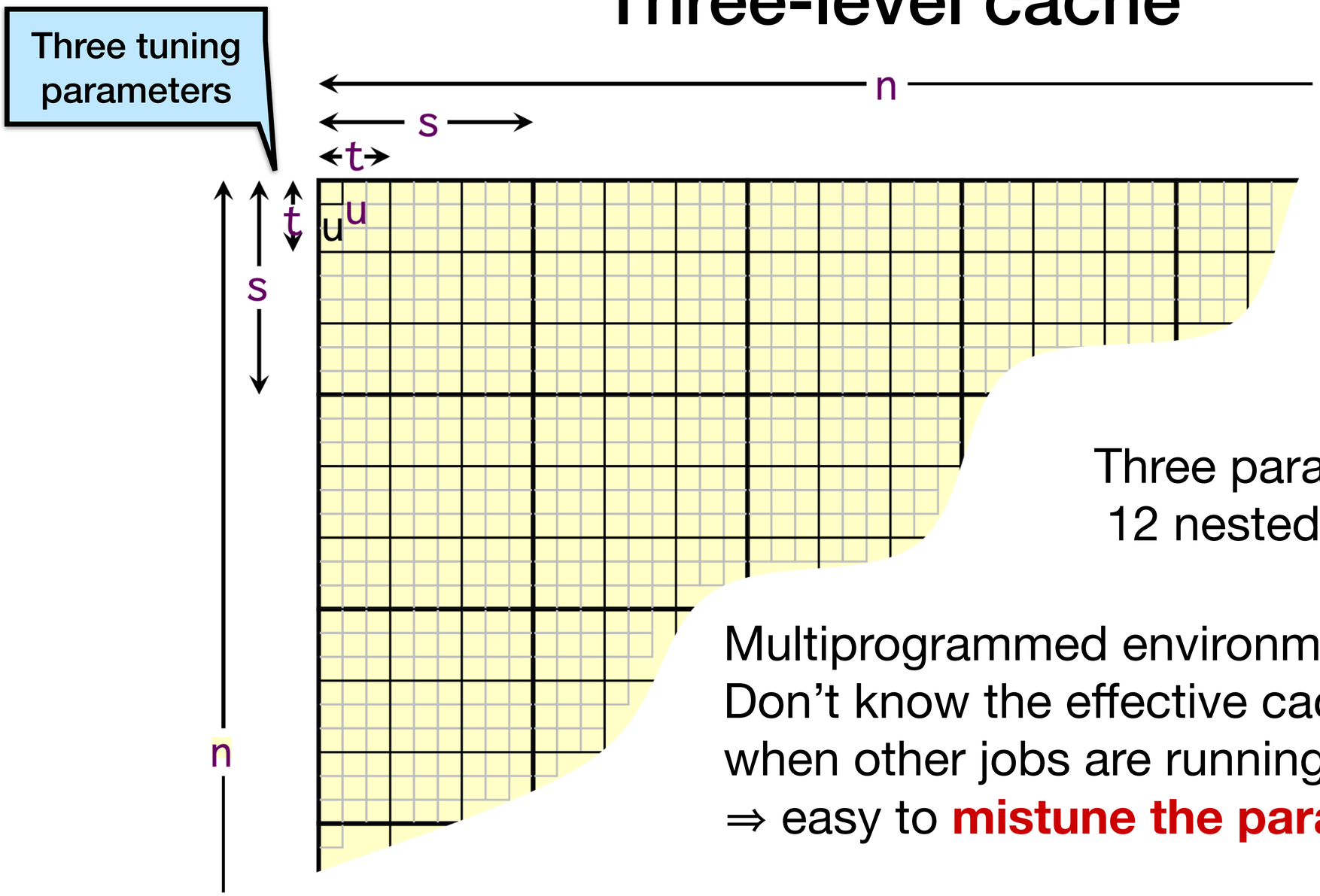
Problem: multidimensional tuning optimization cannot be done with binary search.

Two-level cache




```
void Tiled_Mult2(double *C, double *A, double *B, int64_t n) {
  for (int64_t i2=0; i2<n; i2+=s)
    for (int64_t j2=0; j2<n; j2+=s)
      for (int64_t k2=0; k2<n; k2+=s)
        for (int64_t i1=i2; i1<i2+s && i1<n; i1+=t)
          for (int64_t j1=j2; j1<j2+s && j1<n; j1+=t)
            for (int64_t k1=k2; k1<k2+s && k1<n; k1+=t)
              for (int64_t i=i1; i<i1+s && i<i2+t && i<n; i++)
                for (int64_t j=j1; j<j1+s && j<j2+t && j<n; j++)
                  for (int64_t k=k1; k1<k1+s && k<k2+t && k<n; k++)
                    C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Three-level cache



Three parameters ->
12 nested for loops

Multiprogrammed environment:
Don’t know the effective cache size
when other jobs are running
⇒ easy to **mistune the parameters**!

Divide and Conquer

Recursive matrix multiplication

Divide-and-conquer on nxn matrices:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

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

8 multiply-adds of `(n/2)x(n/2)` matrices.

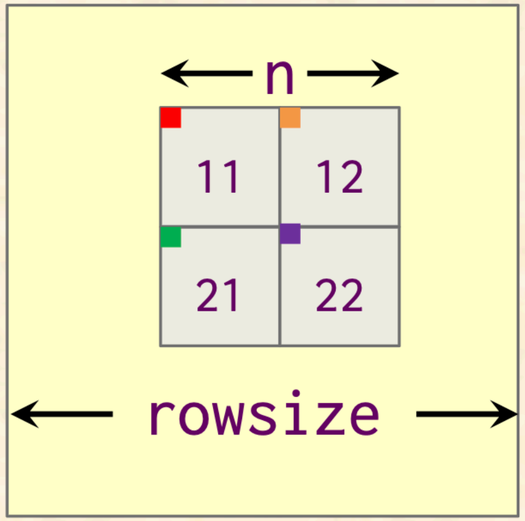
Recursive code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```

Coarsen base case to overcome function-call overheads.

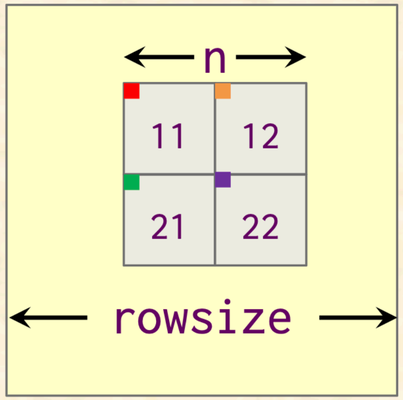
Recursive code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```



Analysis of work

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```



$$W(n) = 8W(n/2) + \Theta(1)$$
$$= \Theta(n^3)$$

Analysis of work

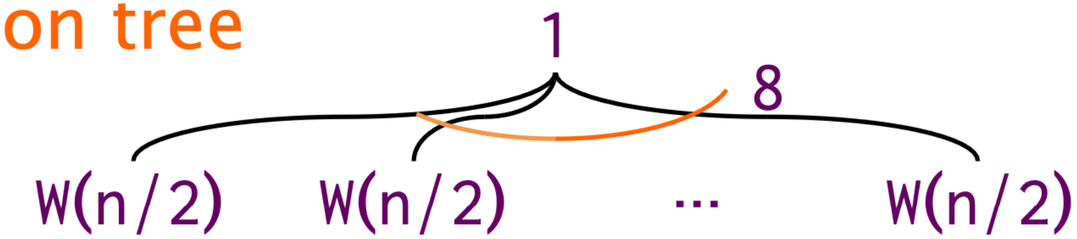
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree W(n)

Analysis of work

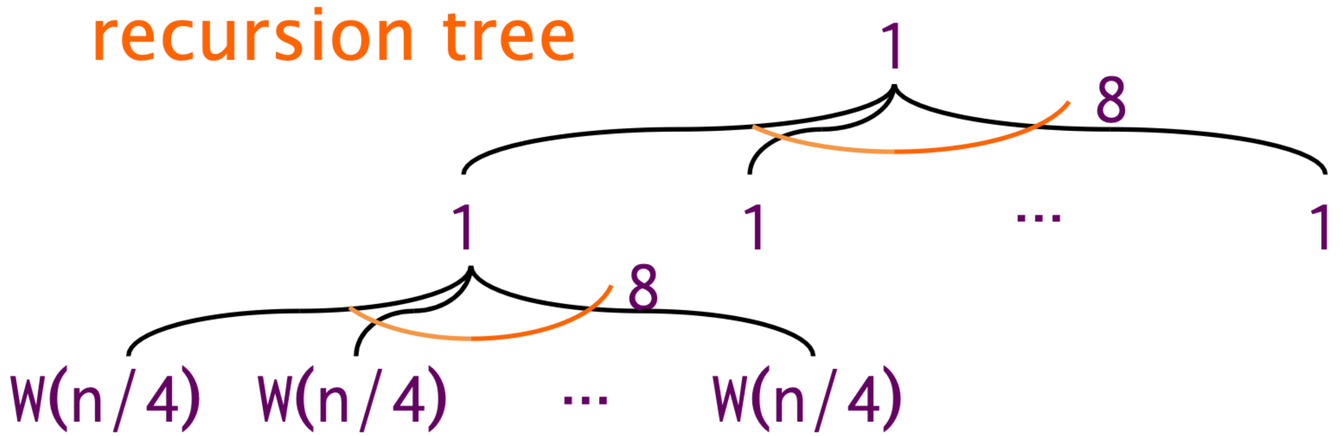
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree



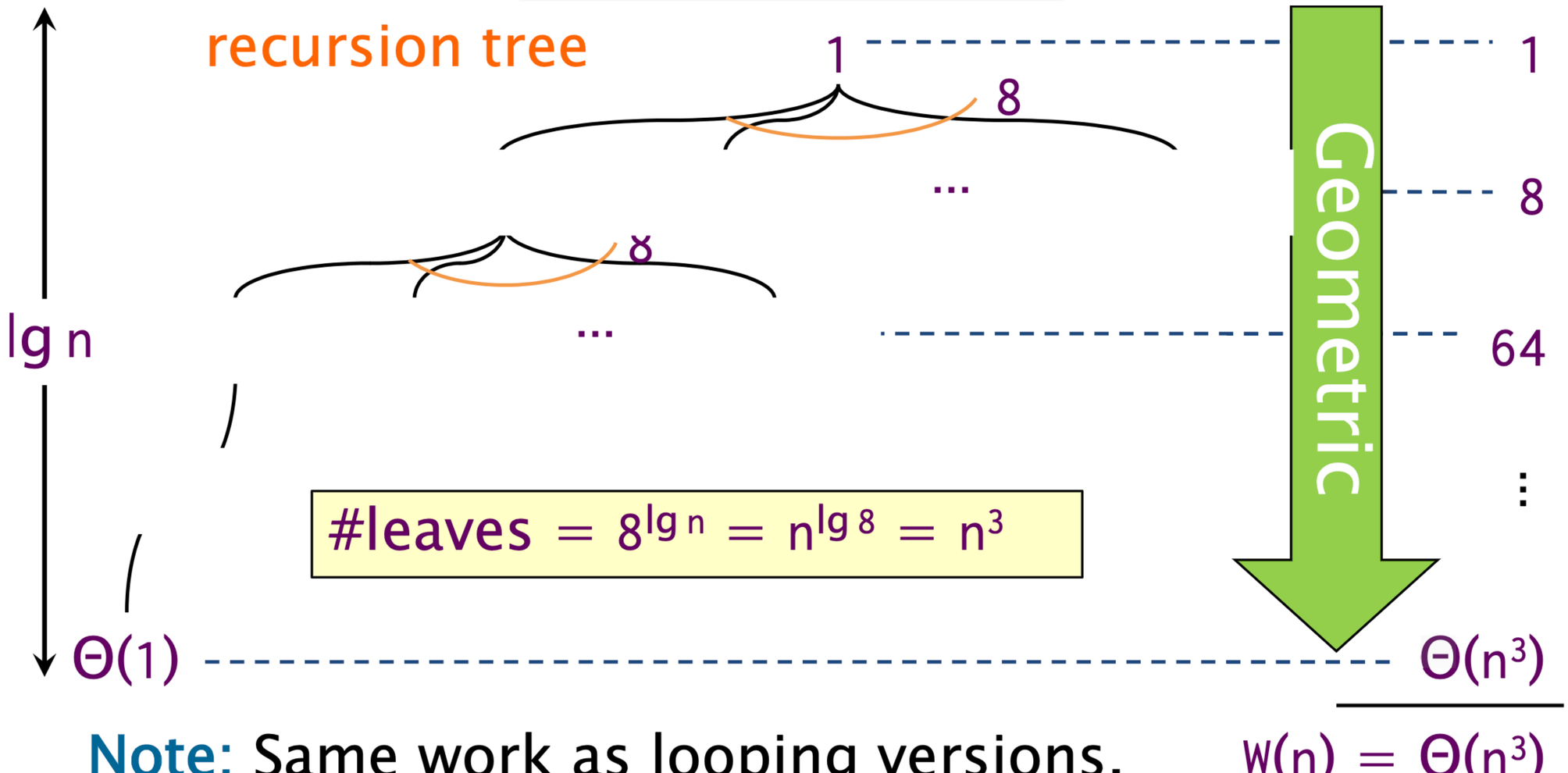
Analysis of work

$$W(n) = 8W(n/2) + \Theta(1)$$



Analysis of work

$$W(n) = 8W(n/2) + \Theta(1)$$



Note: Same work as looping versions.

Analysis of cache misses

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```

Submatrix Caching Lemma

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

Analysis of cache misses

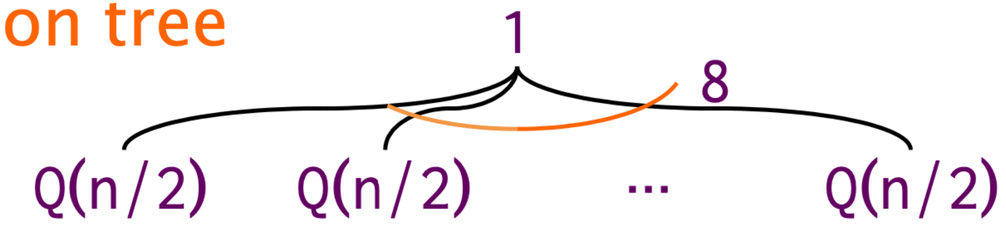
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

recursion tree $Q(n)$

Analysis of cache misses

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

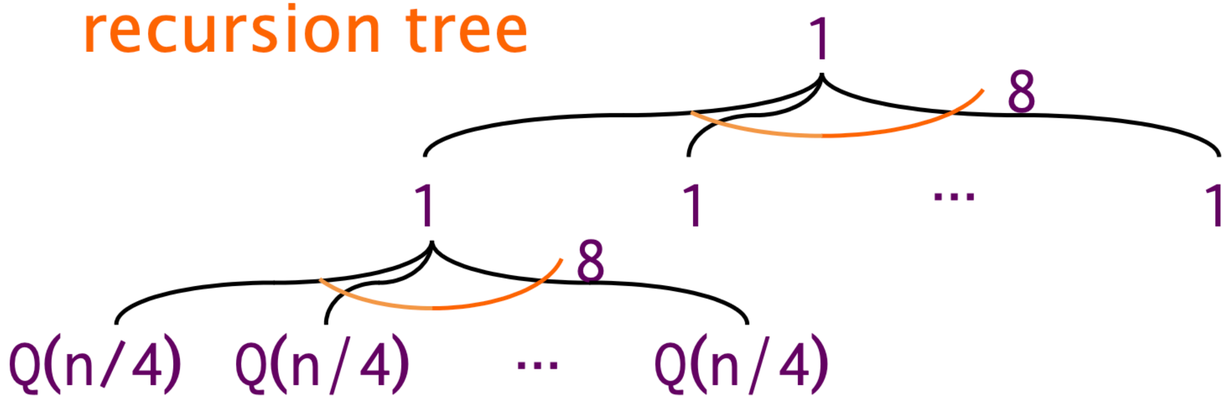
recursion tree



Analysis of cache misses

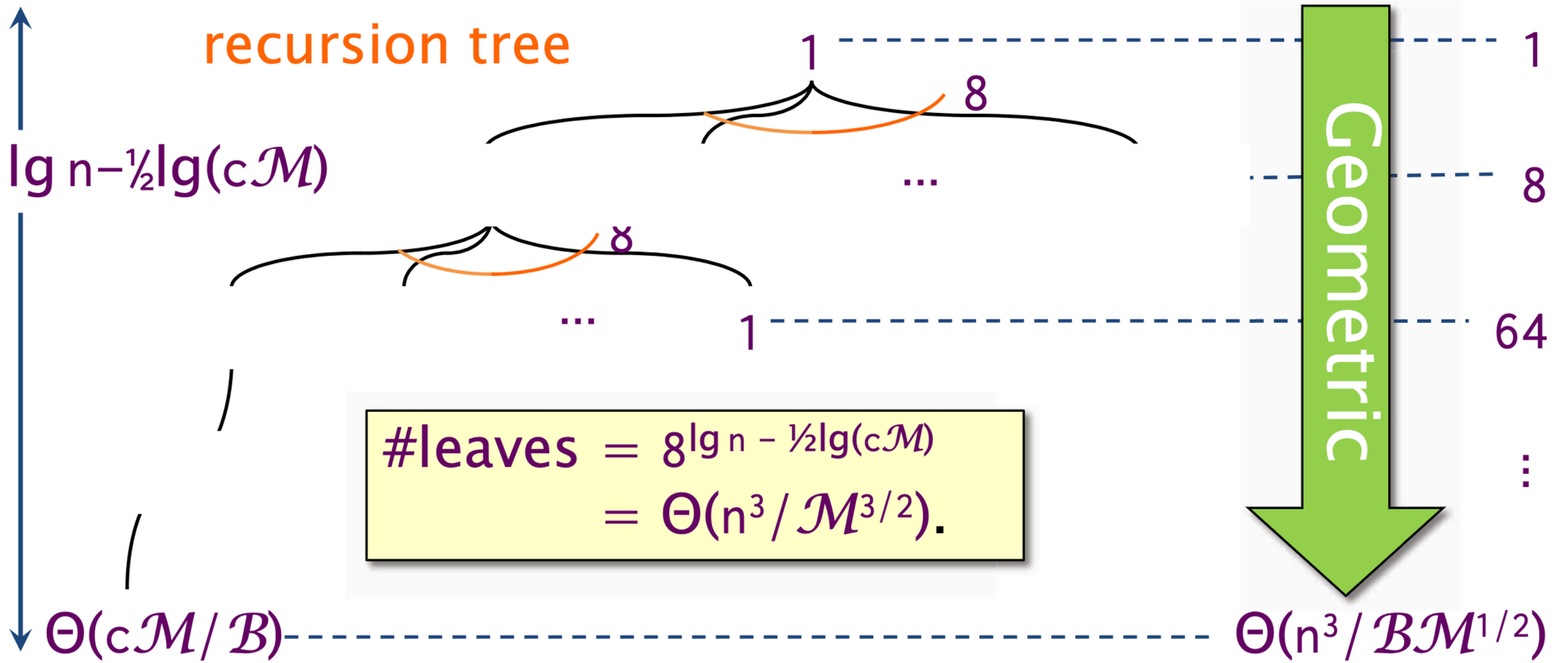
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

recursion tree



Analysis of cache misses

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$



#leaves $= 8^{\lg n - \frac{1}{2}\lg(c\mathcal{M})}$
$= \Theta(n^3/\mathcal{M}^{3/2})$.

$$Q(n) = \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$$

Same cache misses as with tiling!

Efficient cache-oblivious algorithms

- No tuning parameters.
- No explicit knowledge of cache sizes.
- Handle multilevel caches automatically (asymptotically optimally).
- Good in multiprogrammed environments (see: cache-adaptive algorithms).