6.172 Performance Engineering of Software Systems

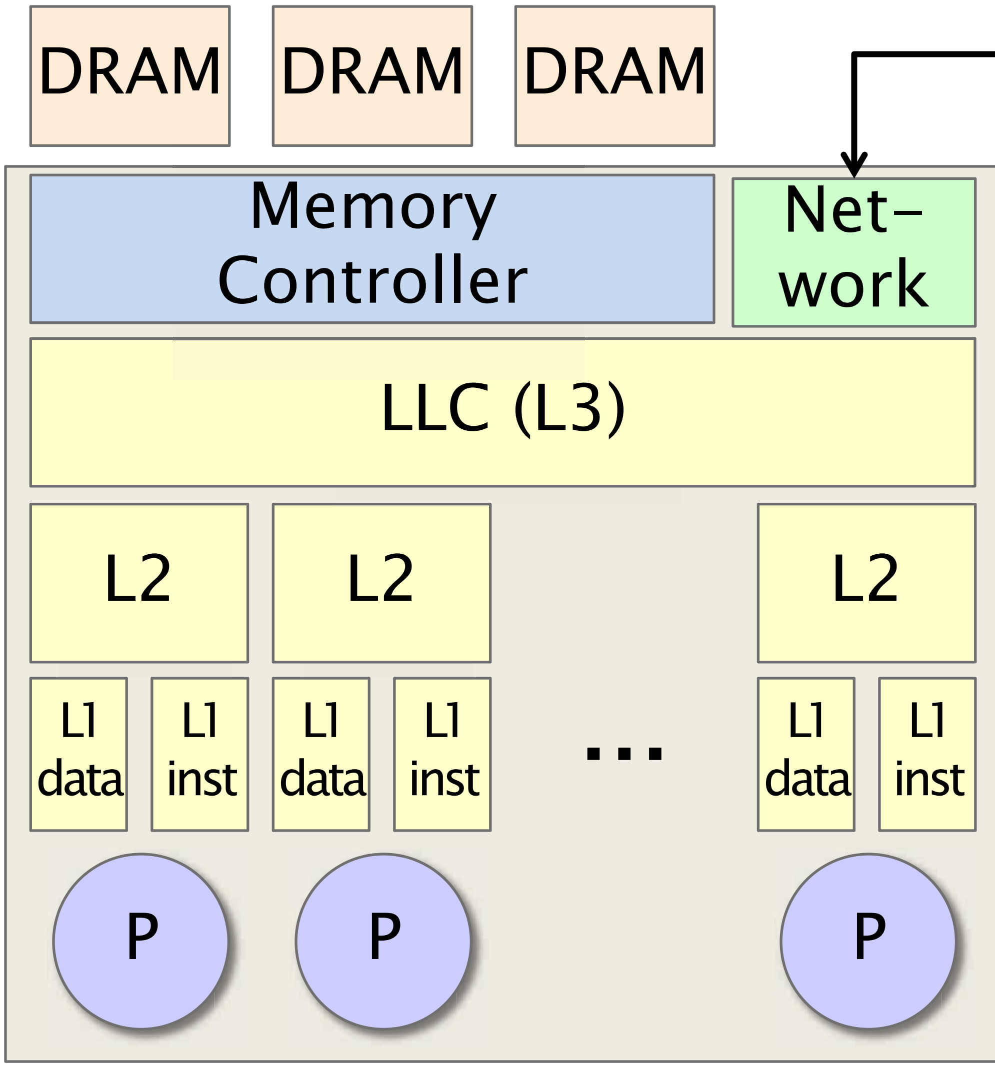
LECTURE 14

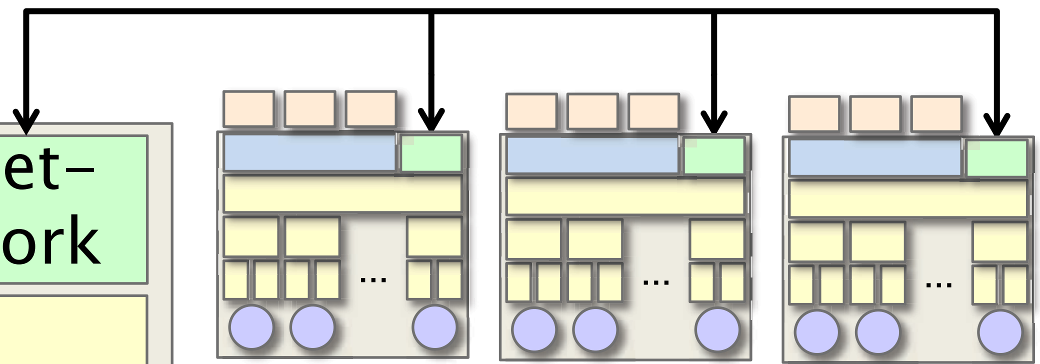
Caching and Cache-Efficient Algorithms

Julian Shun

Cache Hardware

Multicore Cache Hierarchy





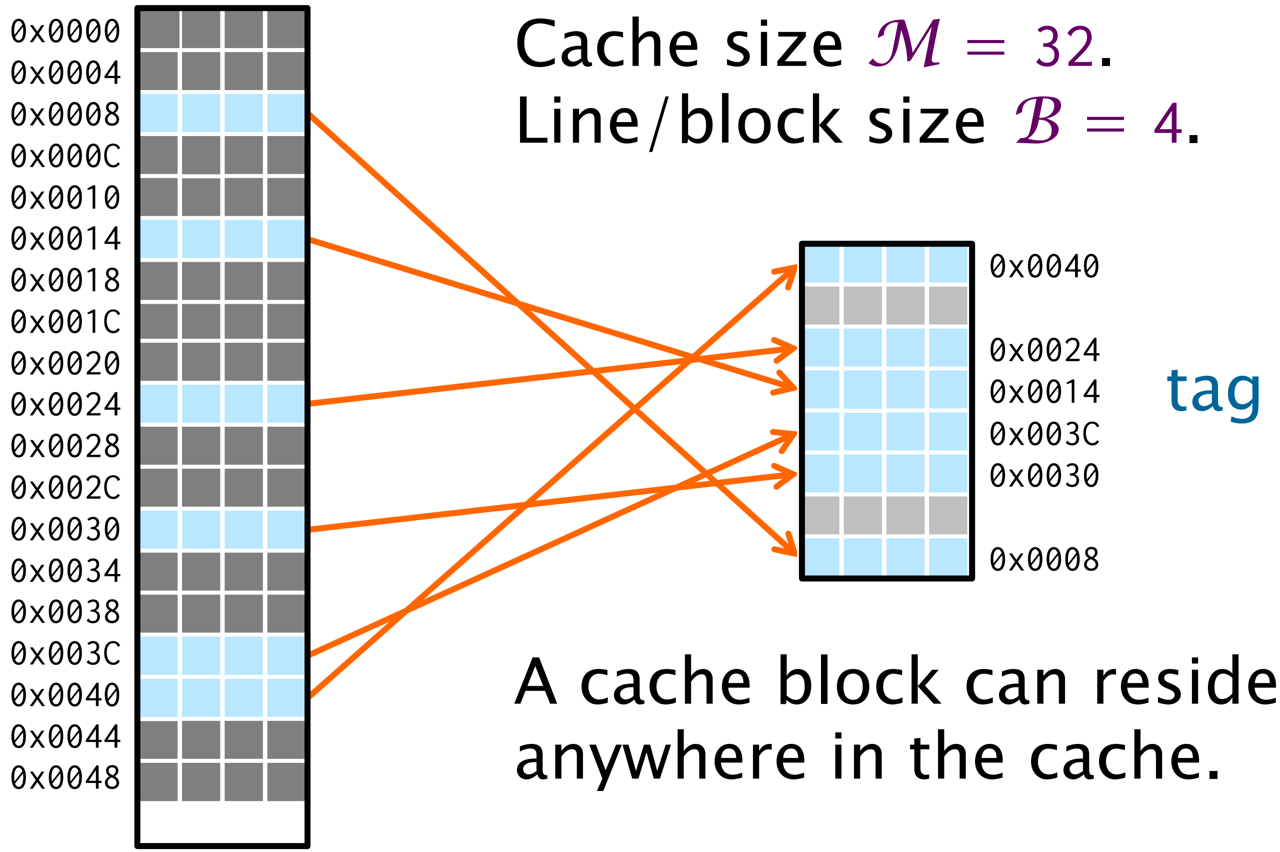
Level	Size	Assoc.	Latency (ns)
Main	128 GB		50
LLC	30 MB	20	6
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64 B cache blocks

Fully Associative Cache

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

w–bit address space

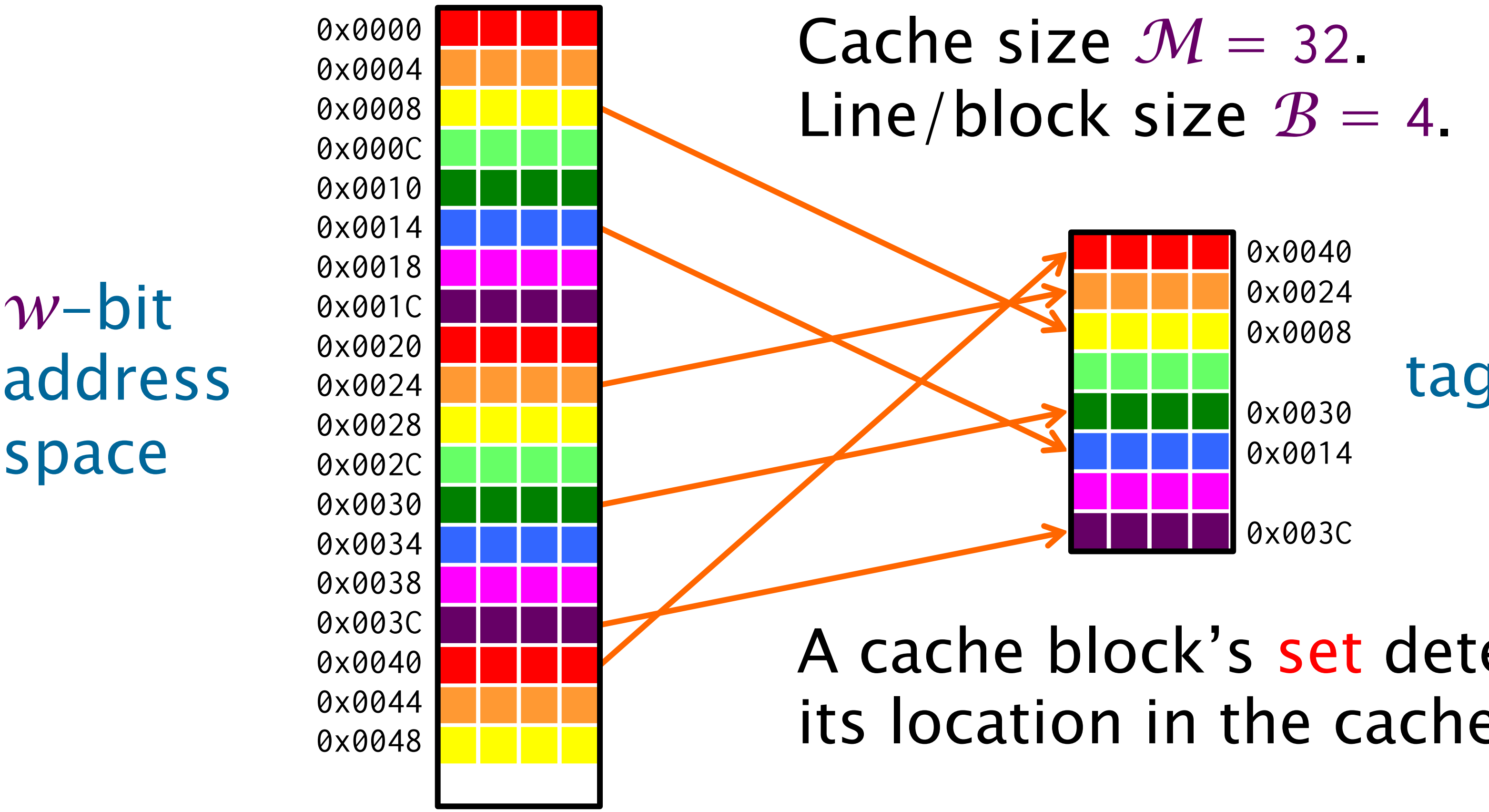


A cache block can reside anywhere in the cache.

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.

Direct-Mapped Cache

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



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

address

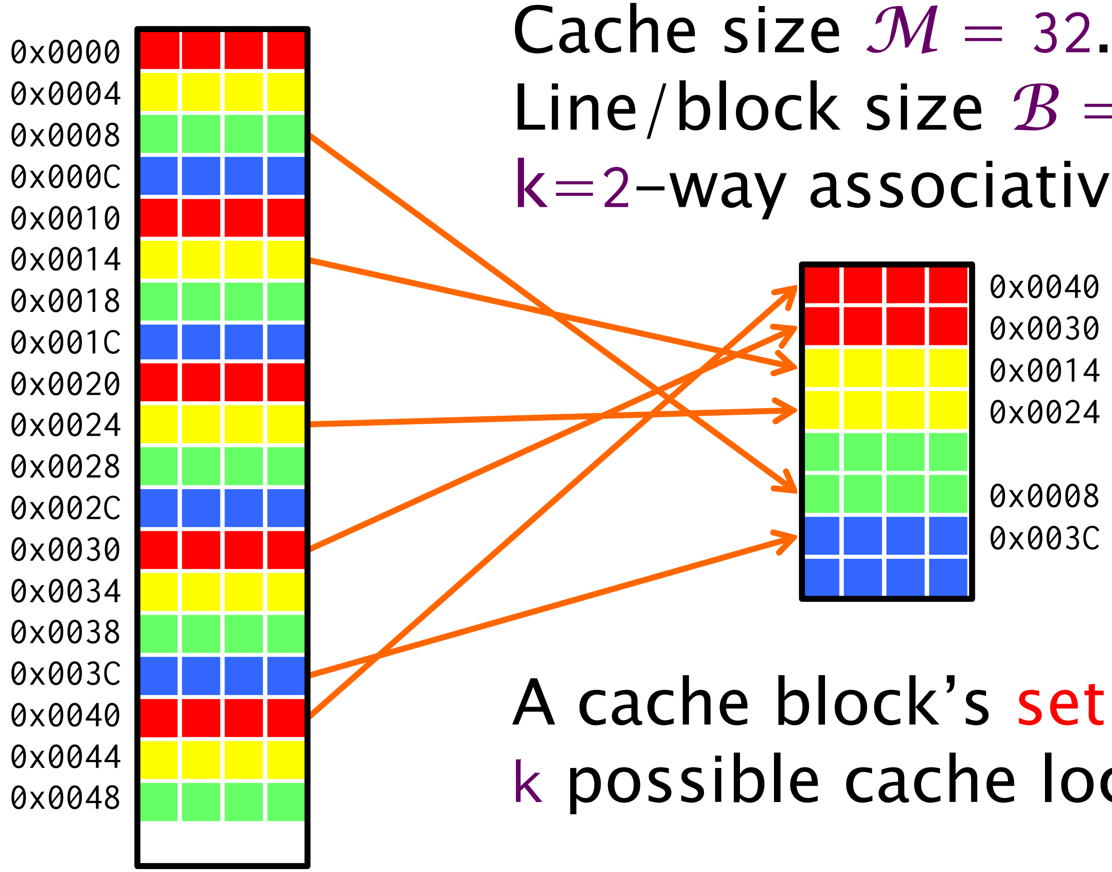
	tag	set	offset
bits	$w - \lg \mathcal{M}$	$\lg(\mathcal{M}/\mathcal{B})$	$\lg \mathcal{B}$

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

Set-Associative Cache

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

w-bit address space



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.

Taxonomy of Cache Misses

Cold miss

- The first time the cache block is accessed.

Capacity miss

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

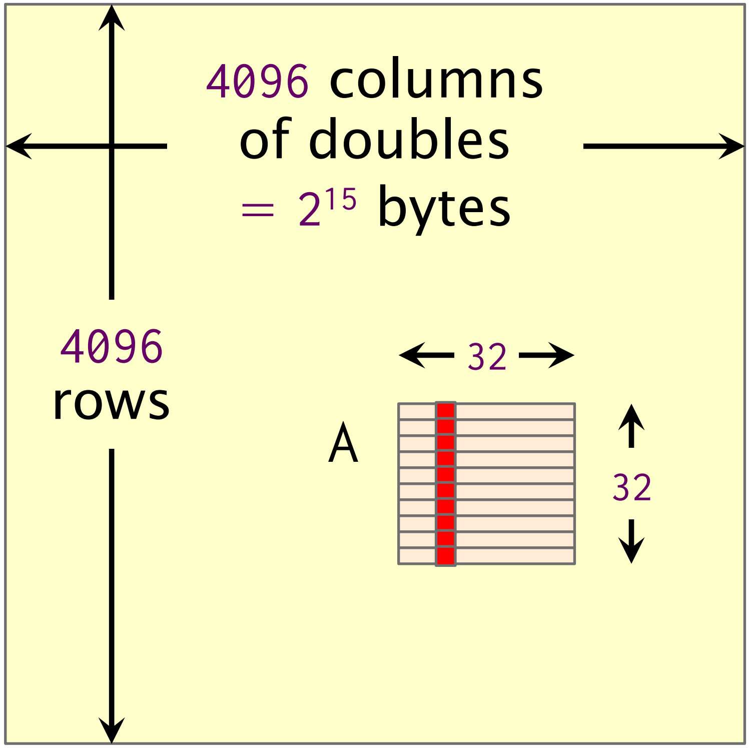
Conflict miss

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

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.

Conflict Misses for Submatrices



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

Assume:

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

Analysis

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

Solutions

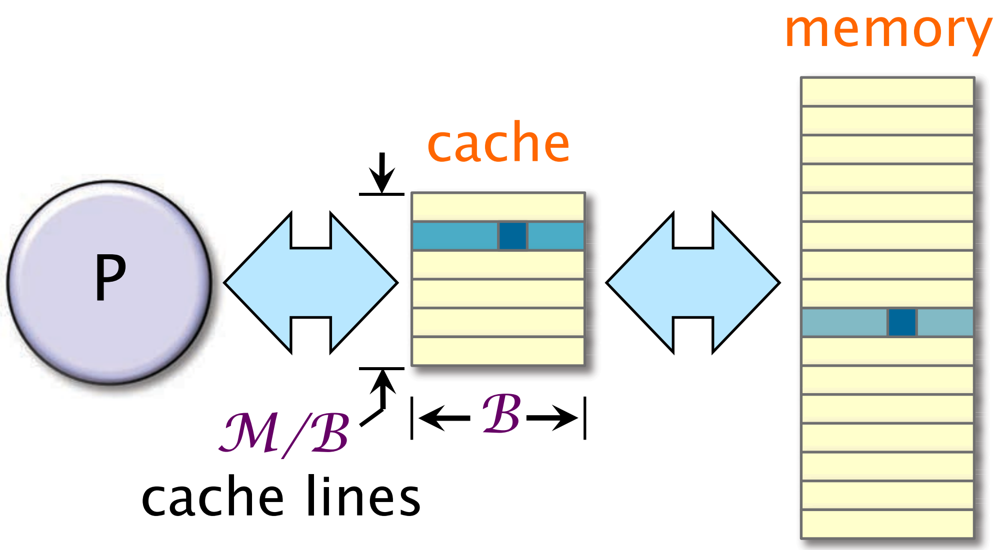
Copy A into a temporary 32×32 matrix, or pad rows.

Ideal-Cache Model

Ideal-Cache Model

Parameters

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



Performance Measures

- work `W` (ordinary running time)
- cache misses `Q`

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

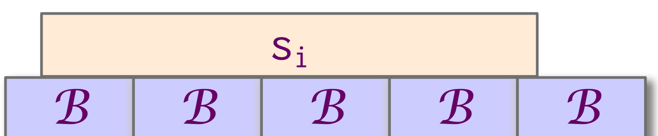
Lemma. Suppose that a program reads a set of r data segments, where the ith segment consists of s_i bytes, and suppose that

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

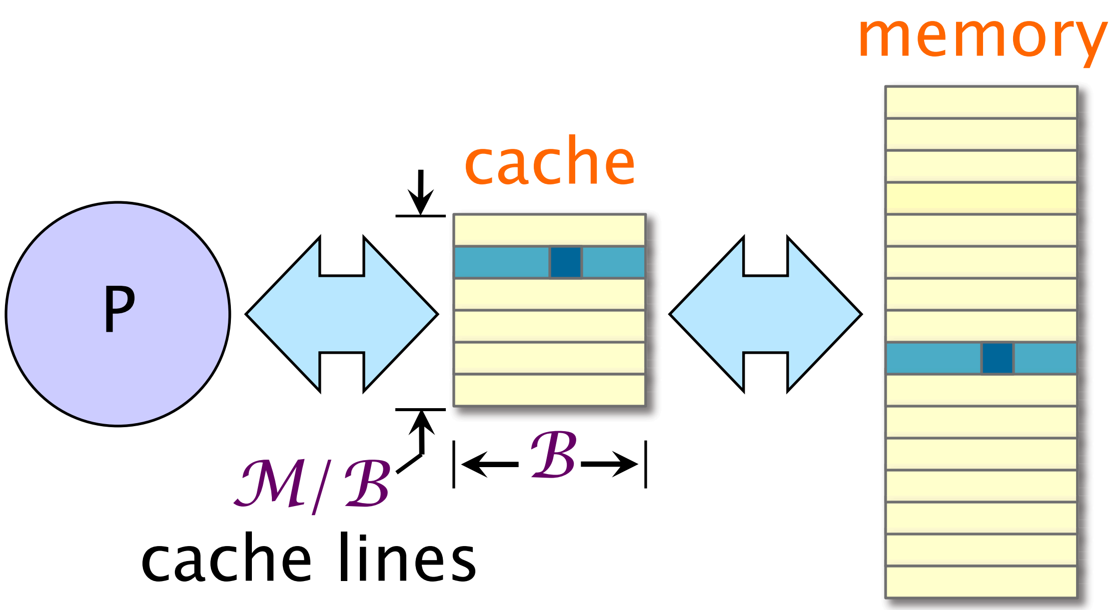
Then all the segments fit into cache, and the number of misses to read them all is at most $3N/\mathcal{B}$.

Proof. A single segment s_i incurs at most $s_i/\mathcal{B} + 2$ misses, and hence we have

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



Tall Caches



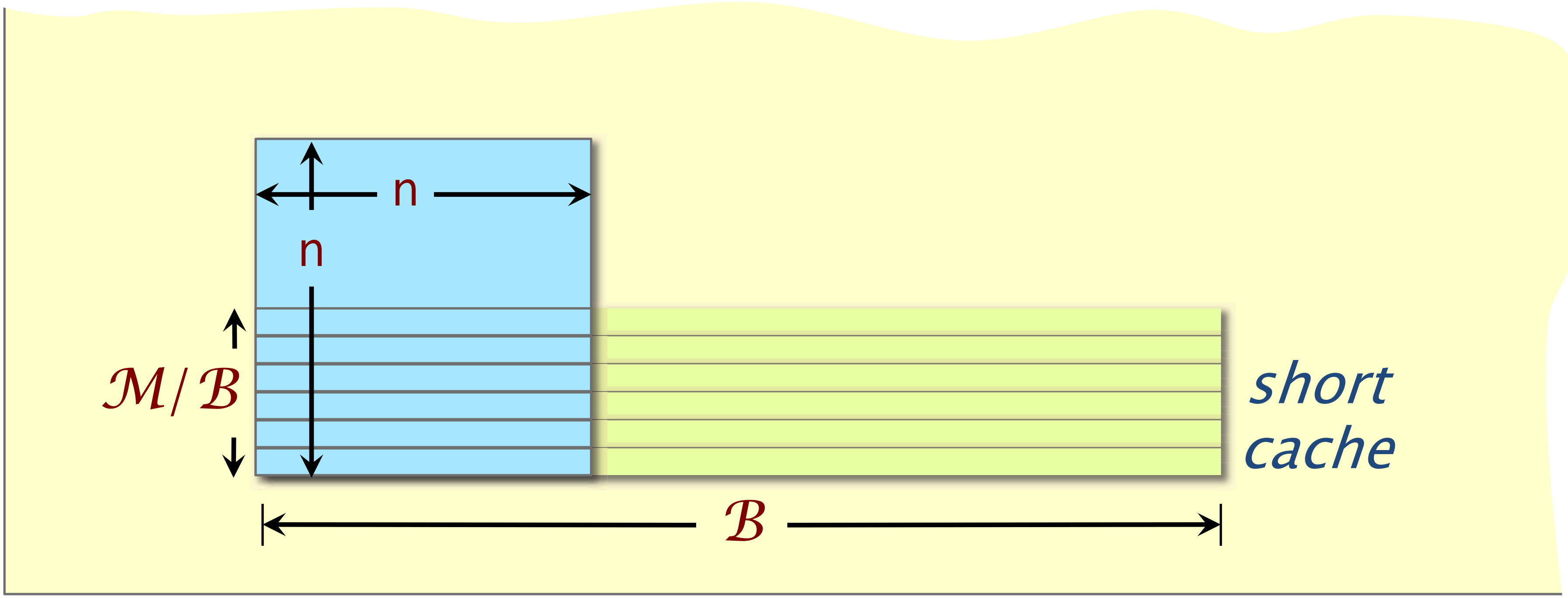
Tall-cache assumption

$\mathcal{B}^2 < c\,\mathcal{M}$ for some sufficiently small constant $c \le 1$.

Example: Intel Xeon E5-2666 v3

- Cache-line length = 64 bytes.
- L1-cache size = 32 Kbytes.

What's Wrong with Short Caches?

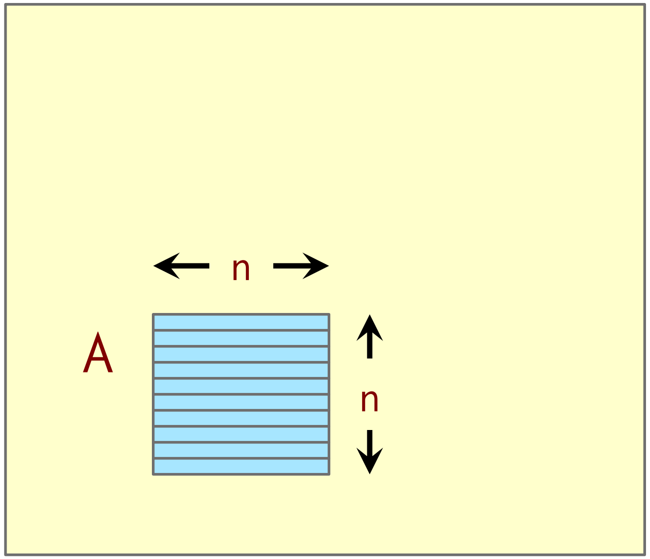


Tall-cache assumption

$\mathcal{B}^2 < c\mathcal{M}$ 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 $c \le 1$ is constant, and suppose that $c\mathcal{M} \le n^2 < \mathcal{M}/3$. 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 $N = n^2$, $n = r = s_i$, $\mathcal{B} \le n = N/r$, and $N < \mathcal{M}/3$. Thus, the Cache-Miss Lemma applies. ■

Cache Analysis of Matrix Multiplication

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];
}
```

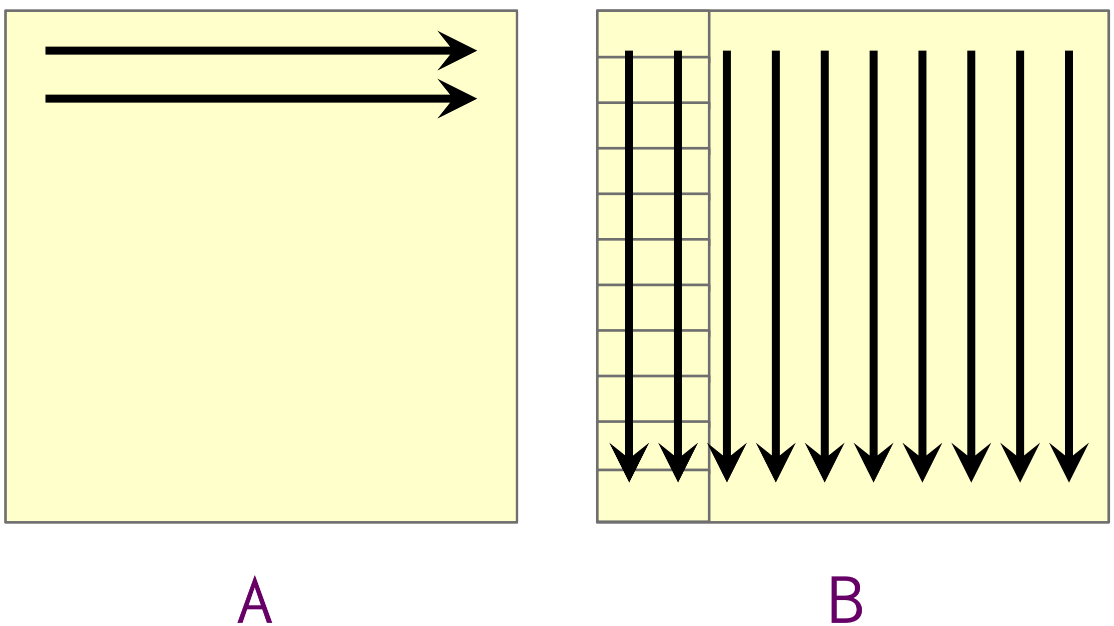
Analysis of work

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

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



Case 1

$n > c\mathcal{M}/\mathcal{B}$.

Analyze matrix B.

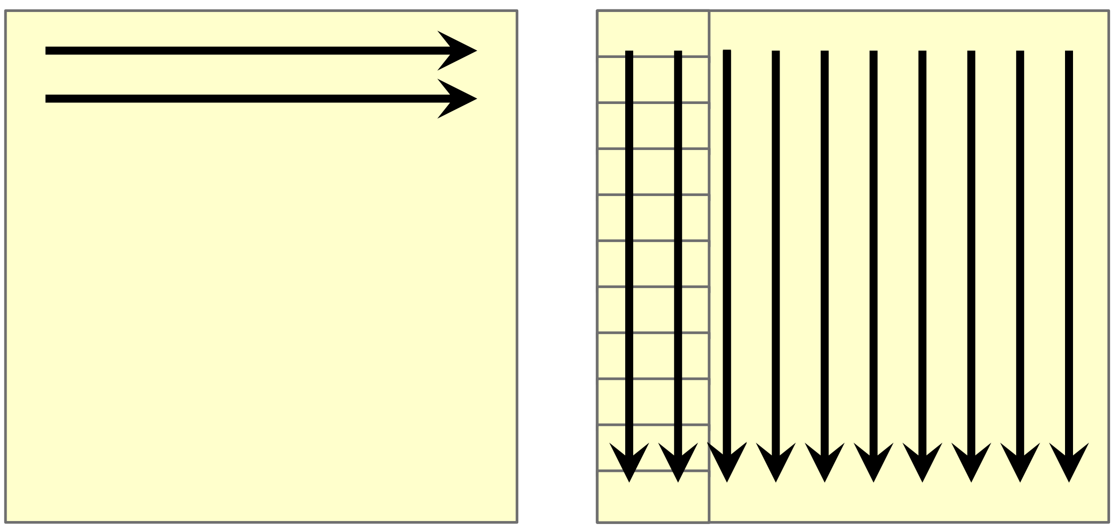
Assume LRU.

$Q(n) = \Theta(n^3)$, 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

$c' \mathcal{M}^{1/2} < n < c\mathcal{M}/\mathcal{B}$.

Analyze matrix B.

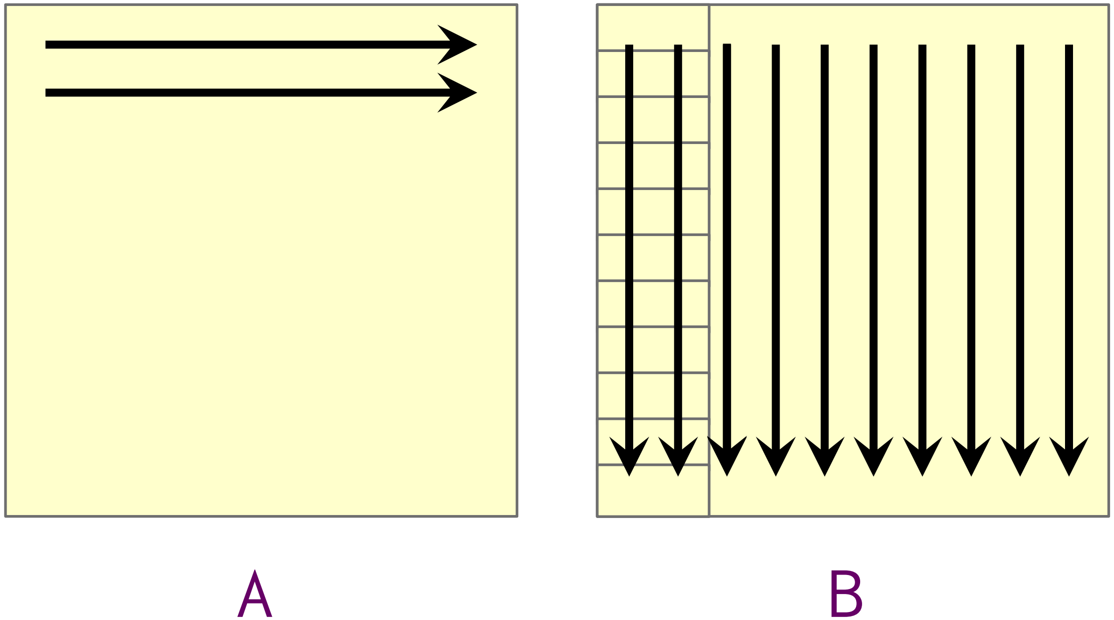
Assume LRU.

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



Case 3

$n < c'\mathcal{M}^{1/2}$.

Analyze matrix B.

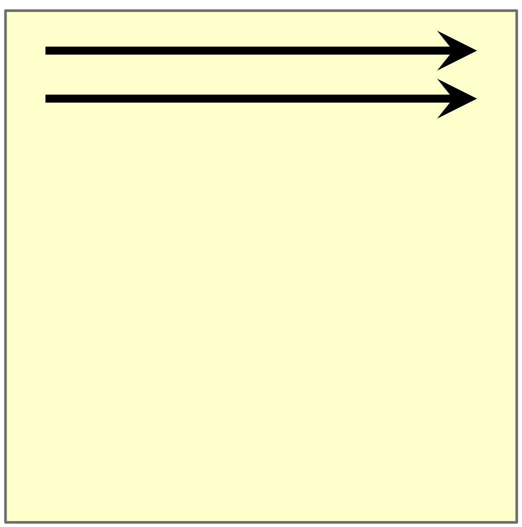
Assume LRU.

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

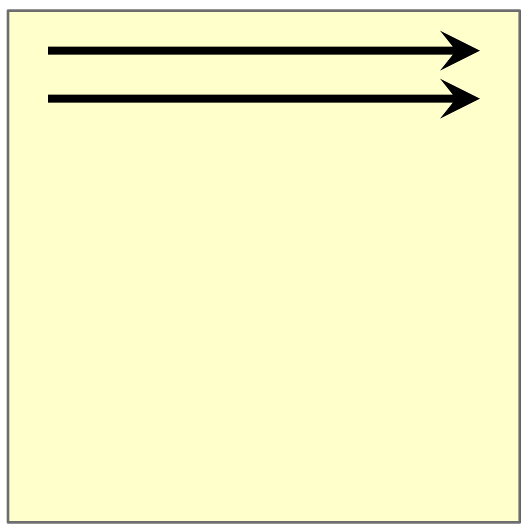
Swapping Inner Loop Order

```
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



C



B

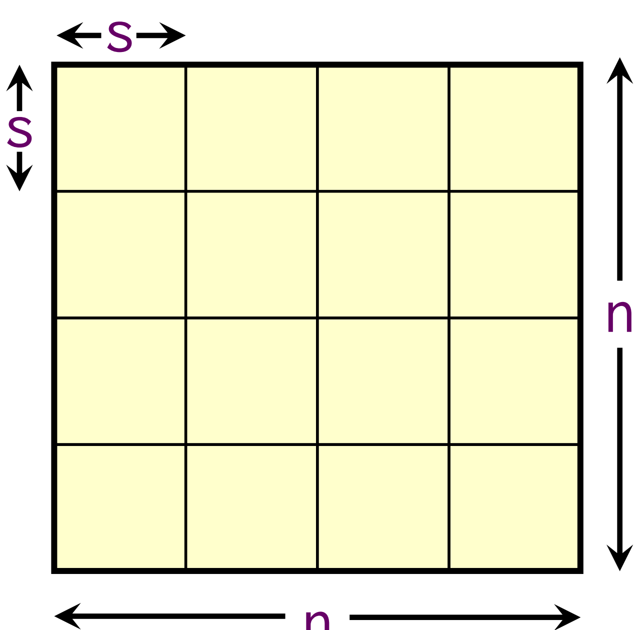
Analyze matrix B.
Assume LRU.

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

Tiling

Tiled Matrix Multiplication

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

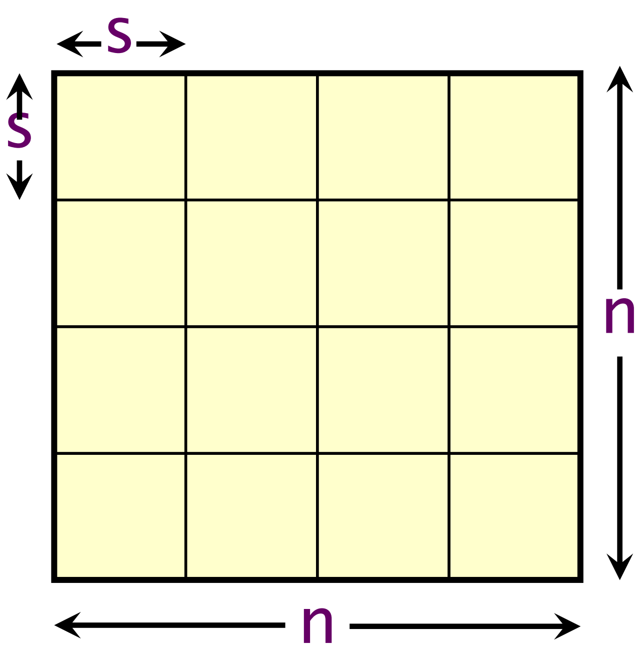


Analysis of work

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

Tiled Matrix Multiplication

```
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];
}
```



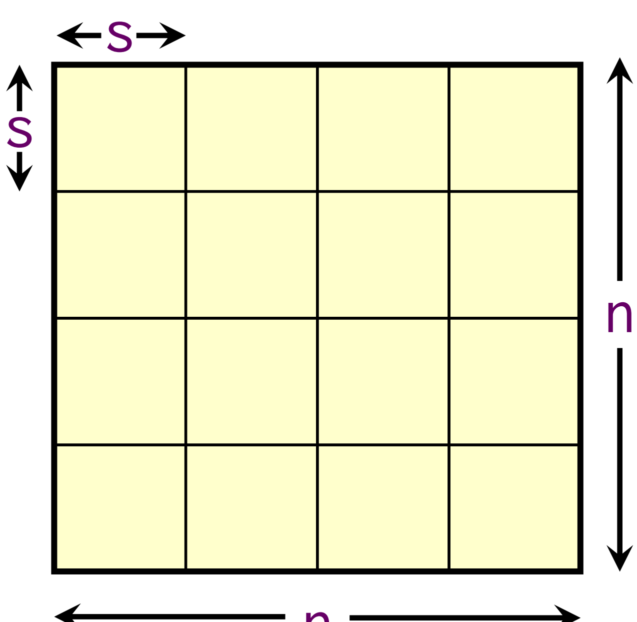
Analysis of cache misses

- Tune s so that the submatrices just fit into cache $\Rightarrow s = \Theta(\mathcal{M}^{1/2})$.
- Submatrix Caching Lemma implies $\Theta(s^2/\mathcal{B})$ misses per submatrix.
- $Q(n) = \Theta((n/s)^3(s^2/\mathcal{B}))$
  $= \boxed{\Theta(n^3/(\mathcal{B}\mathcal{M}^{1/2}))}$. *Remember this!*
- Optimal [HK81].

Tiled Matrix Multiplication

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1
    for (int
      for (i
        for
          for (
            for (i
              C[i*n+j] += A[i
}
```

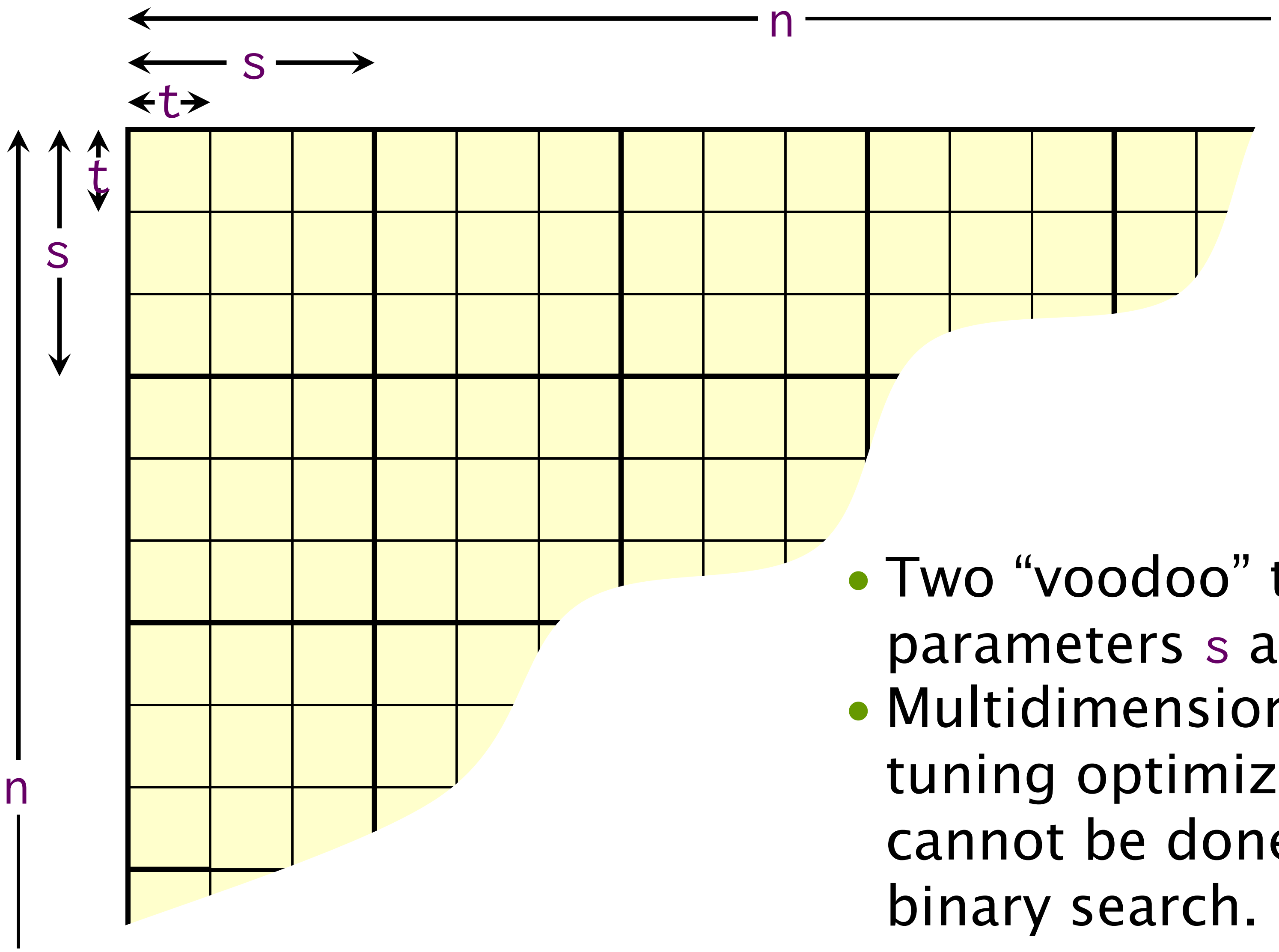
Voodoo!



Analysis of cache misses

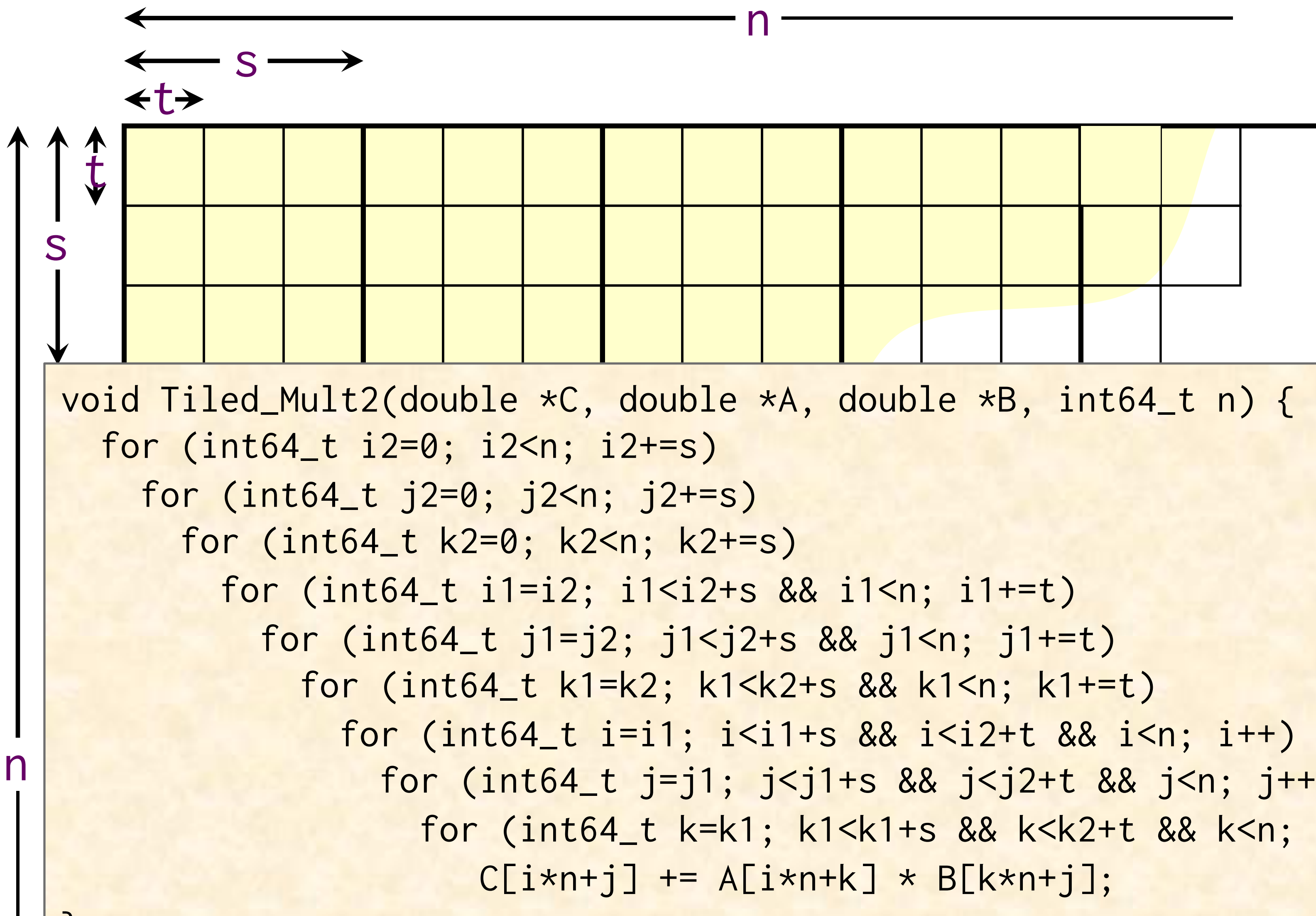
- Tune s so that the submatrices just fit into cache $\Rightarrow s = \Theta(\mathcal{M}^{1/2})$.
- Submatrix Caching Lemma implies $\Theta(s^2/\mathcal{B})$ misses per submatrix.
- $Q(n) = \Theta((n/s)^3(s^2/\mathcal{B}))$ $= \boxed{\Theta(n^3/(\mathcal{B}\mathcal{M}^{1/2}))}$. *Remember this!*
- Optimal [HK81].

Two-Level Cache



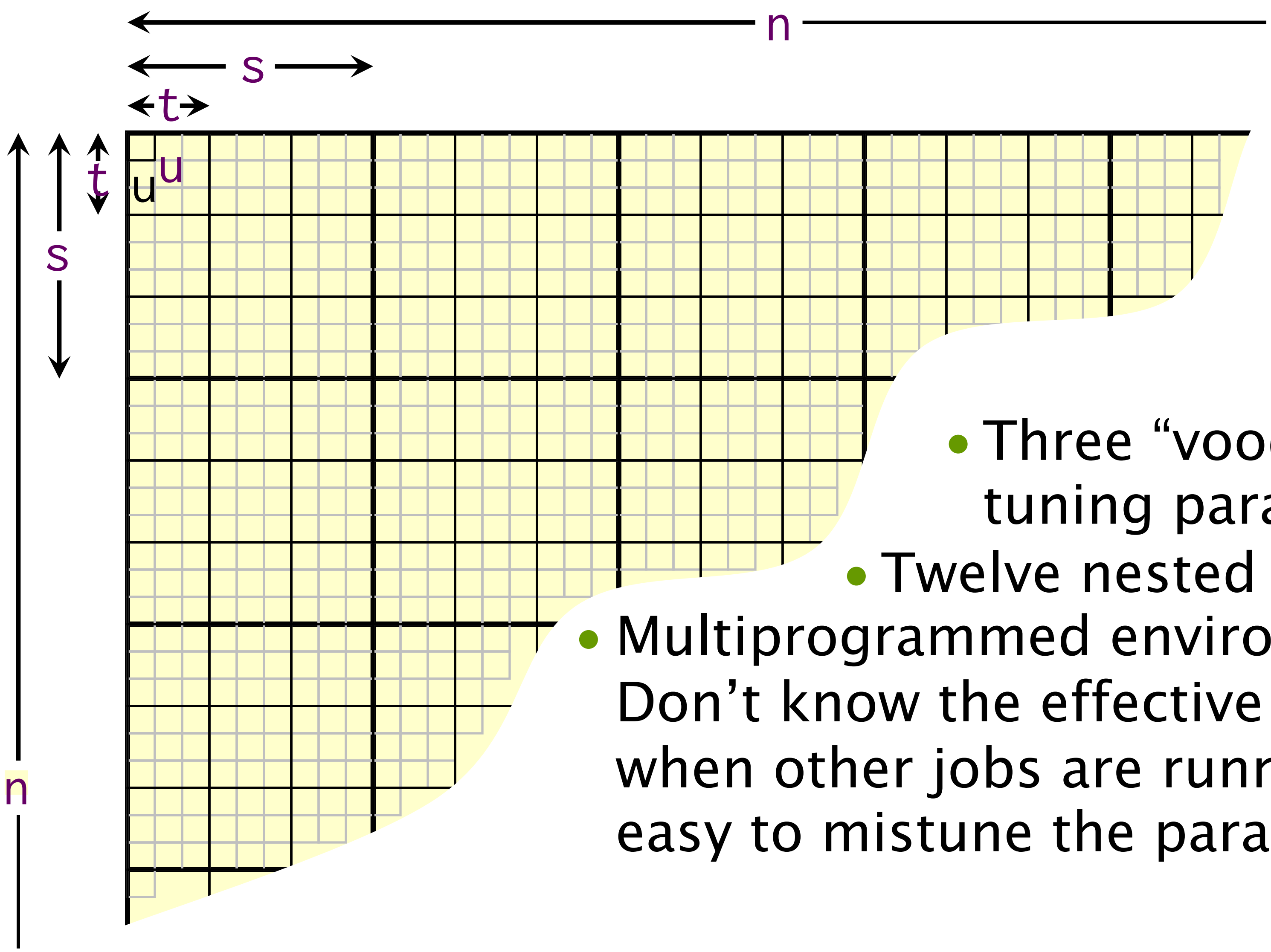
- Two "voodoo" tuning parameters s and t.
- Multidimensional tuning optimization cannot be done with binary search.

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 "voodoo" tuning parameters.
- Twelve nested `for` loops.
- Multiprogrammed environment: Don't know the effective cache size when other jobs are running ⇒ easy to mistune the parameters!

Divide & Conquer

Recursive Matrix Multiplication

Divide-and-conquer on $n \times n$ 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 $(n/2) \times (n/2)$ matrices.

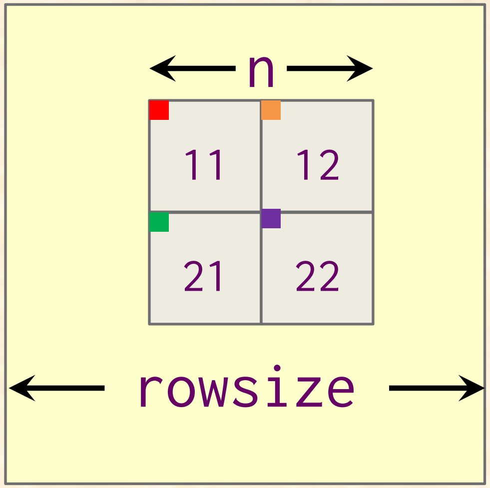
Recursive Code

```
// 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);
} }
```

$$\begin{aligned} W(n) &= 8W(n/2) + \Theta(1) \\ &= \Theta(n^3) \end{aligned}$$

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

$$1$$

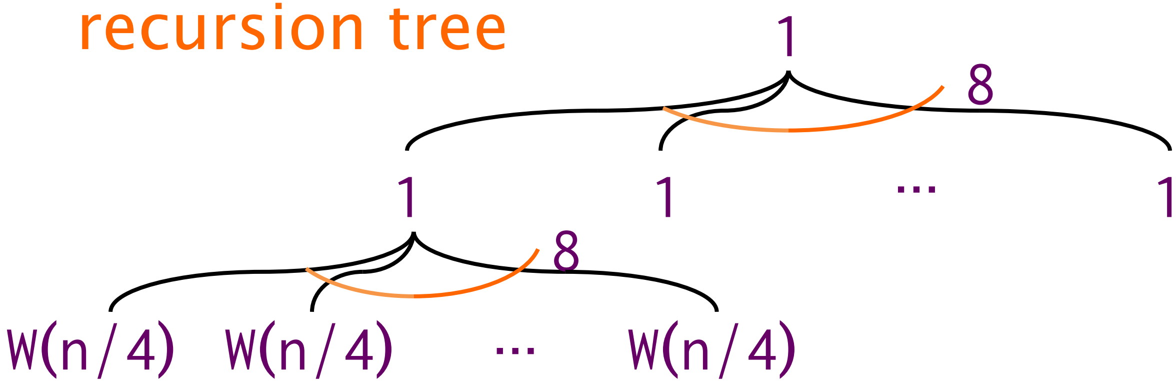
$$8$$

$$W(n/2) \quad W(n/2) \quad \cdots \quad W(n/2)$$

Analysis of Work

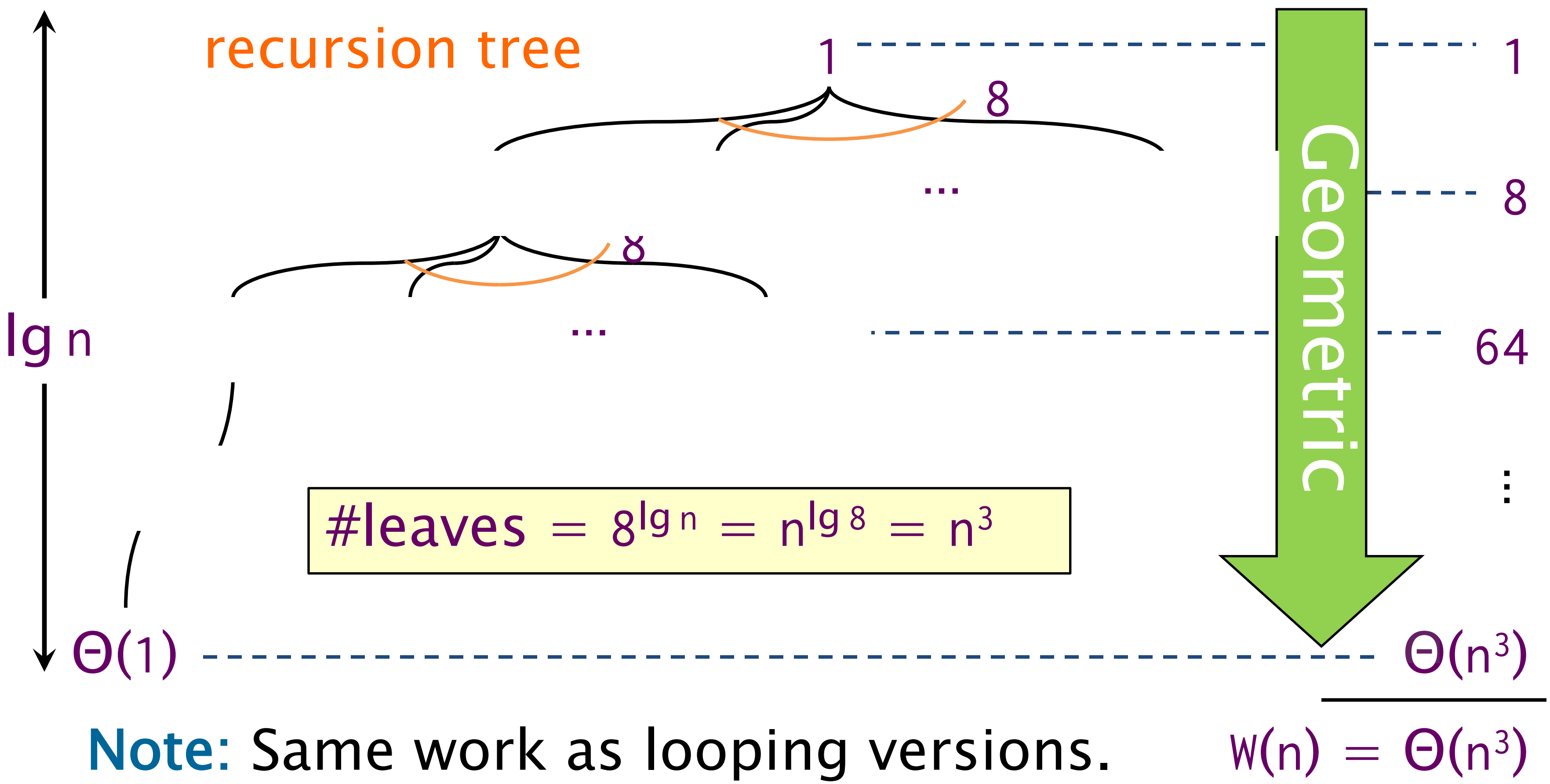
$$\boxed{W(n) = 8W(n/2) + \Theta(1)}$$

recursion tree



Analysis of Work

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



$$\#\text{leaves} = 8^{\lg n} = n^{\lg 8} = n^3$$

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

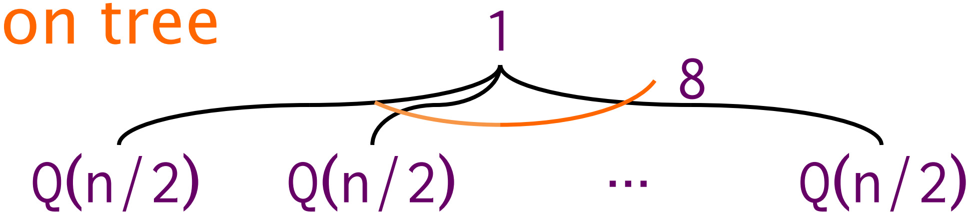
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

recursion tree $\qquad Q(n)$

Analysis of Cache Misses

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

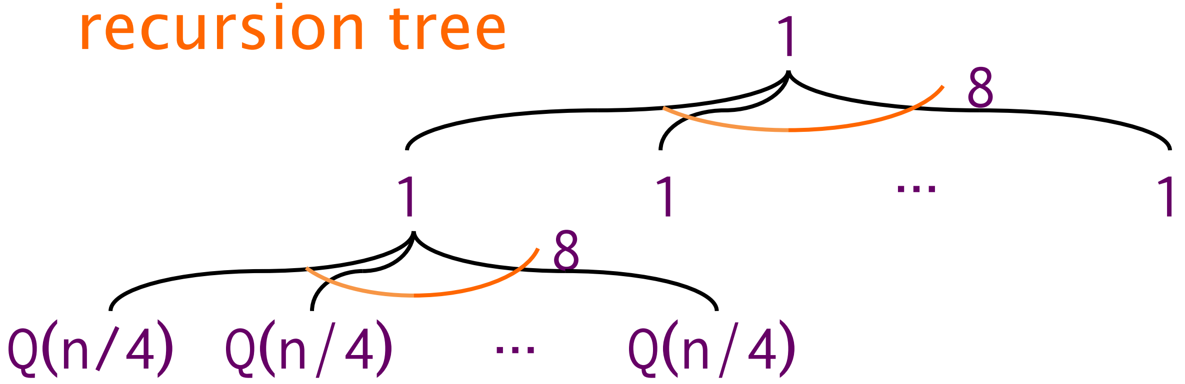
recursion tree



Analysis of Cache Misses

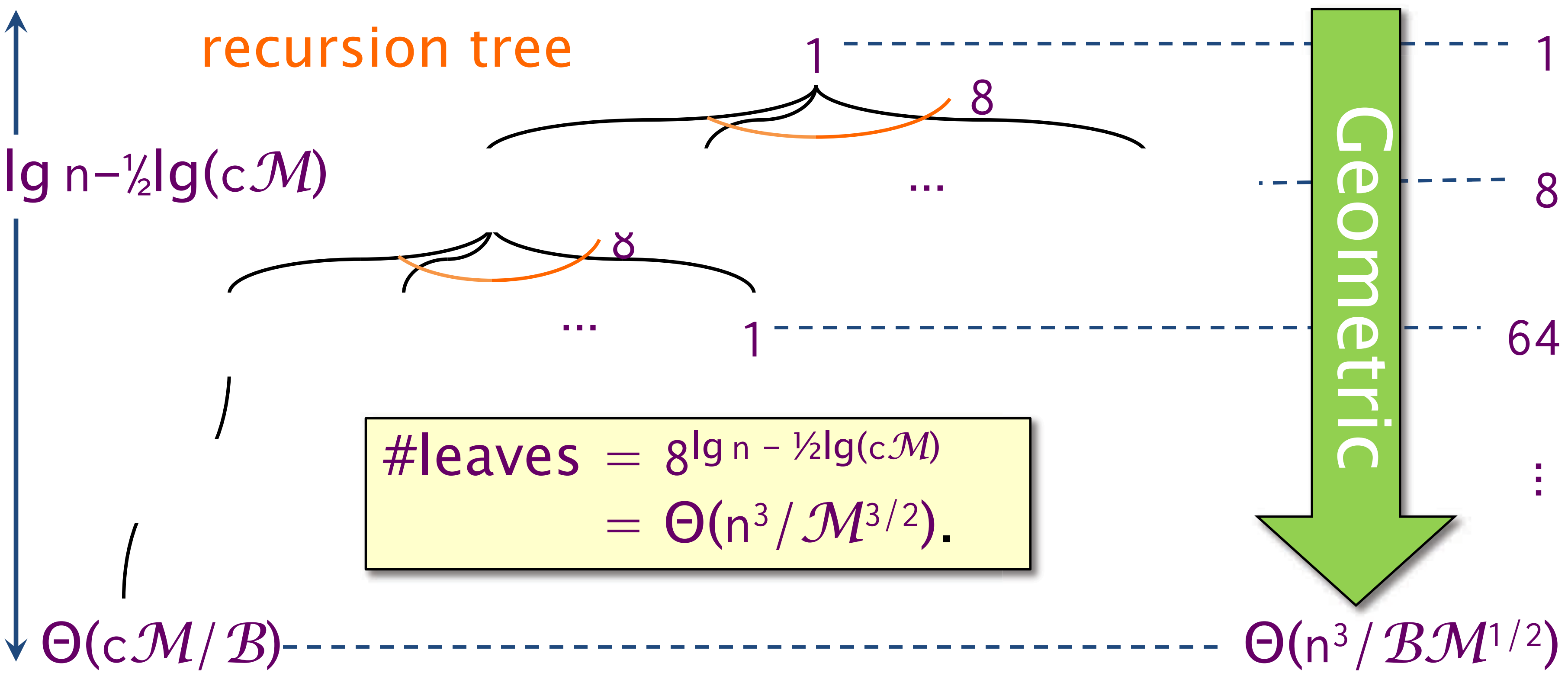
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) & \text{if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

recursion tree



Analysis of Cache Misses

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



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

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

Same cache misses as with tiling!

Efficient Cache-Oblivious Algorithms

- No voodoo tuning parameters.
- No explicit knowledge of caches.
- Passively autotune.
- Handle multilevel caches automatically.
- Good in multiprogrammed environments.

Matrix multiplication
The best cache-oblivious codes to date work on arbitrary rectangular matrices and perform binary splitting (instead of `8`-way) on the largest of `i`, `j`, and `k`.

Recursive Parallel Matrix Multiply

```
// 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);

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

Cilk and Caching

Theorem. Let Q_P be the number of cache misses in a deterministic Cilk computation when run on P processors, each with a private cache of size $\mathcal{M}$, and let S_P be the number of successful steals during the computation. In the ideal-cache model, we have

$$Q_P = Q_1 + O(S_P\mathcal{M}/\mathcal{B}) \ ,$$

where $\mathcal{M}$ is the cache size and $\mathcal{B}$ is the size of a cache block.

Proof. After a worker steals a continuation, its cache is completely cold in the worst case. But after $\mathcal{M}/\mathcal{B}$ (cold) cache misses, its cache is identical to that in the serial execution. The same is true when a worker resumes a stolen subcomputation after a `cilk_sync`. The number of times these two situations can occur is at most $2S_P$. ■

$S_P = O(PT_\infty)$ in expectation

MORAL: Minimizing cache misses in the serial elision essentially minimizes them in parallel executions.

Recursive Parallel Matrix Multiply

```
// 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);

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

Span: $T_\infty(n) = 2T_\infty(n/2) + \Theta(1) = \Theta(n)$

Cache misses: $Q_P = Q_1 + O(S_P \mathcal{M}/\mathcal{B})$
$= \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2}) + O(Pn\mathcal{M}/\mathcal{B})$

Summary

- Associativity in caches
- Ideal cache model
- Cache-aware algorithms
  - Tiled matrix multiplication
- Cache-oblivious algorithms
  - Divide-and-conquer matrix multiplication
- Cache efficiency analysis in Homework 8