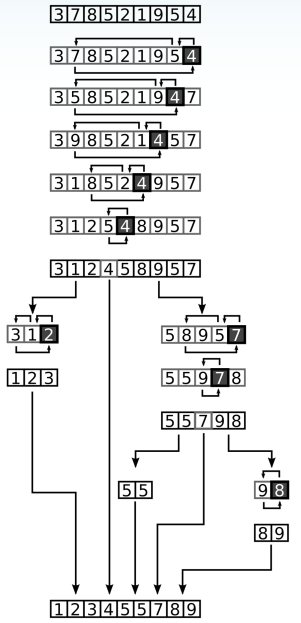
6.506: Algorithm Engineering

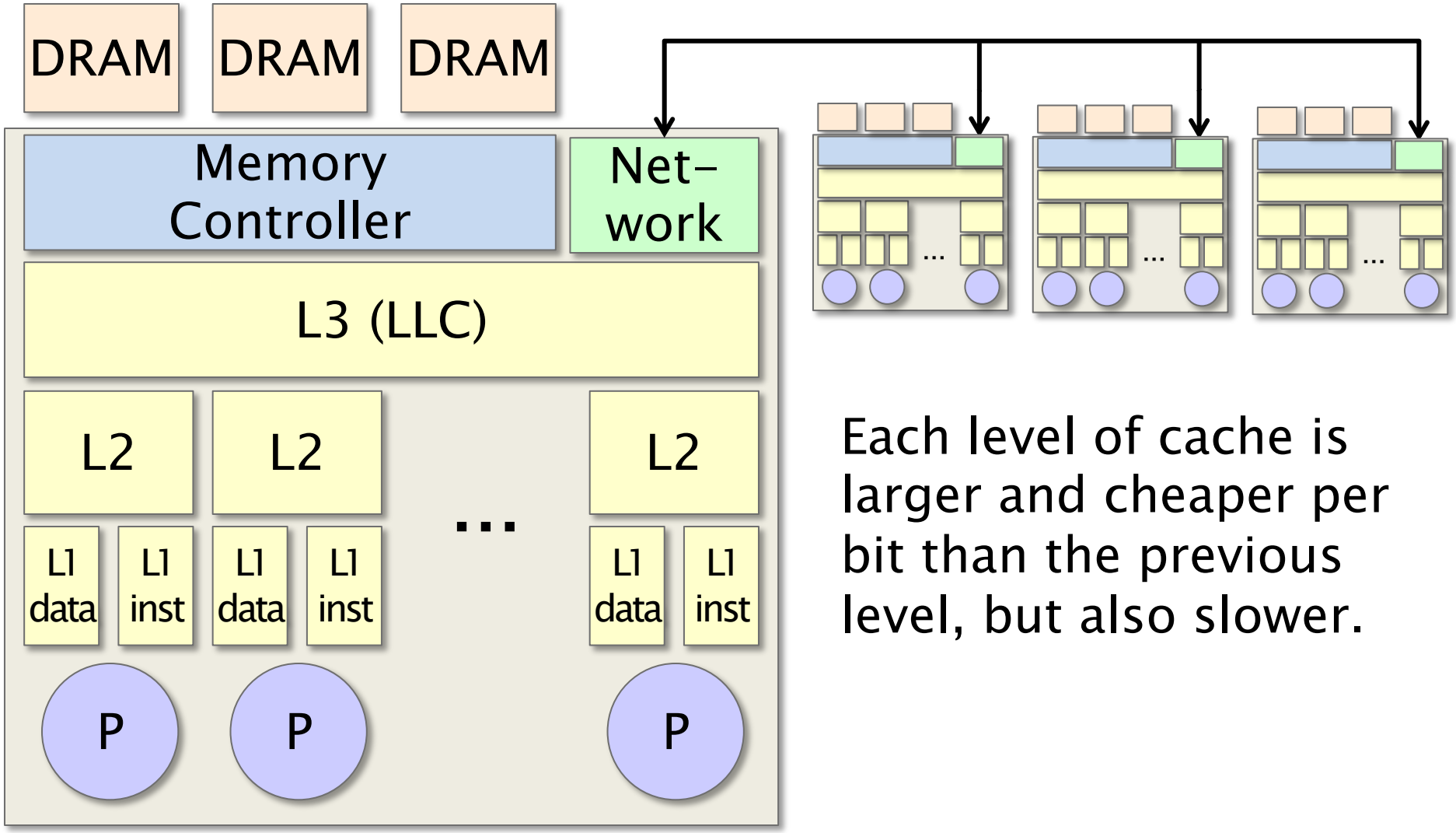


Lecture 3
Cache-Oblivious Algorithms and Data Structures

Charles E. Leiserson

February 14, 2023

Multicore Cache Hierarchy



Each level of cache is larger and cheaper per bit than the previous level, but also slower.

Cache Specs for Typical High-End Multicore

Level	Size/core	Associativity	Latency (cycles)
DRAM	up to 160 GiB		85–240
L3	1.375 MiB	11	50–70
L2	1 MiB	16	14
L1-D	32 KiB	8	4–5
L1-I	32 KiB	8	5

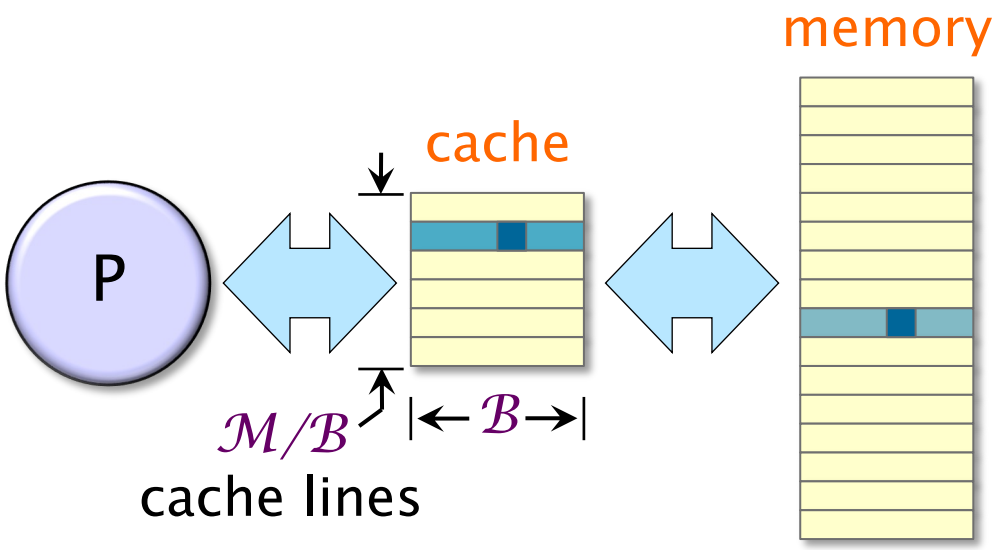
Intel Xeon Platinum 8280L (Cascade Lake)

- Launched April 2019 for $17,906 — cheaper now.
- 2.7 GHz clock, Turbo Boost up to 4 GHz
- 28 cores/chip + 2-way hyperthreading
- 2190 GFLOPS
- 64 B cache lines/blocks
- Up to 8-way multiprocessing

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.

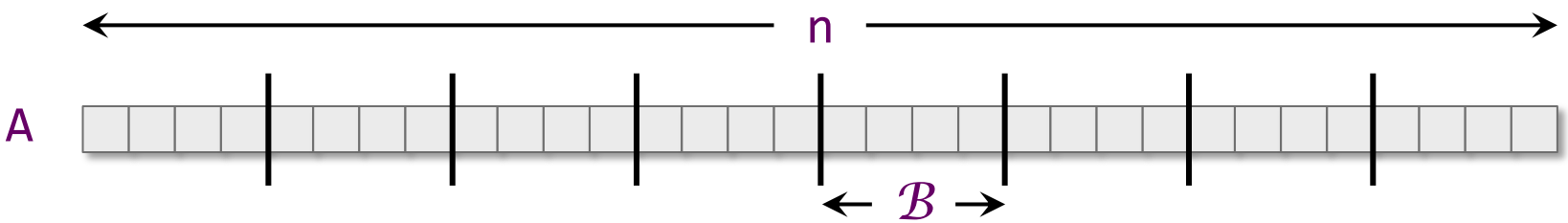


Performance Measures

- work T (ordinary running time)
- cache misses Q

Reading an Array Sequentially

```
sum = 0;
for (int i=0; i<n; ++i)
  sum += A[i];
```



Cache misses: $Q(n) = \Theta(n/\mathcal{B})$

Segment Caching Lemma

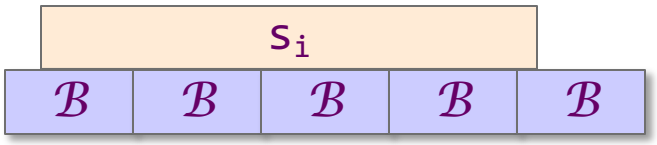
Lemma. Suppose that a program reads a set of r data segments, where the ith segment consists of s_i contiguous bytes in memory, and suppose that

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

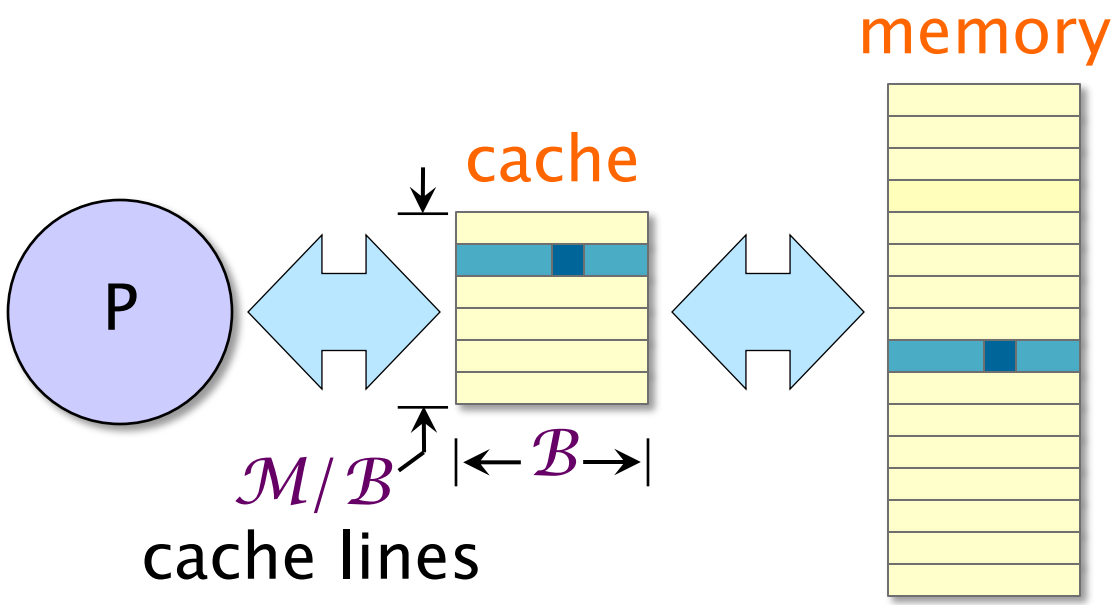
Then all the segments fit into cache, and the number of misses to read them all is at most $3N/\mathcal{B}$.

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

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



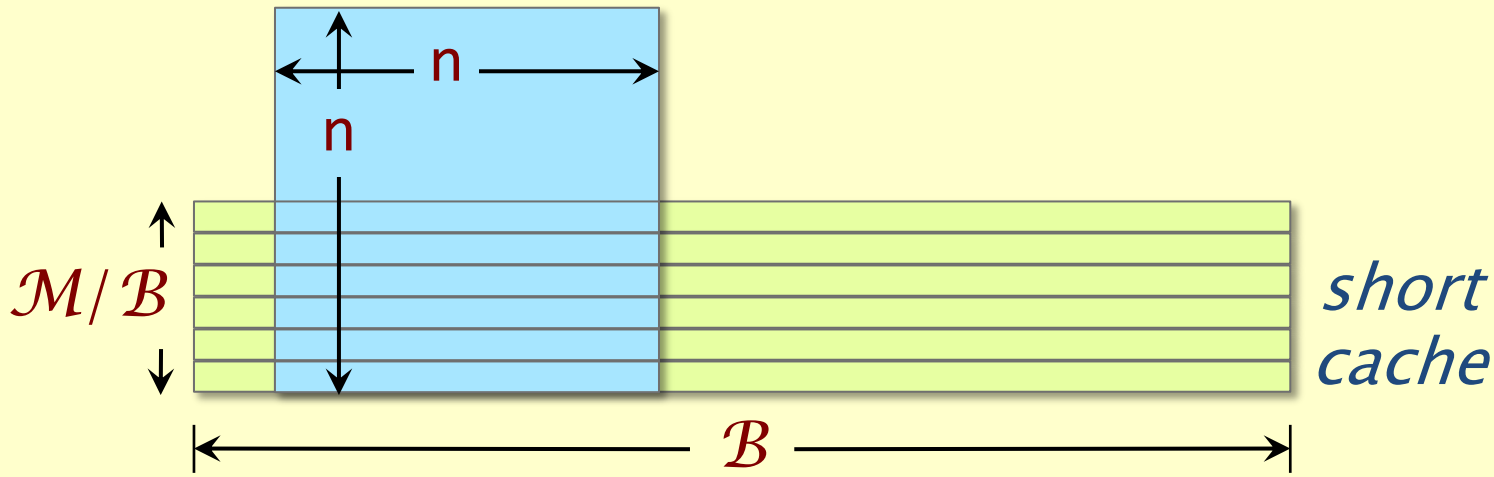
Tall Caches



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

Example: Intel Xeon Platinum 8280L
- Cache-line length $\mathcal{B} = 64$ bytes.
- L1-cache size $\mathcal{M} = 32$ kibibytes.

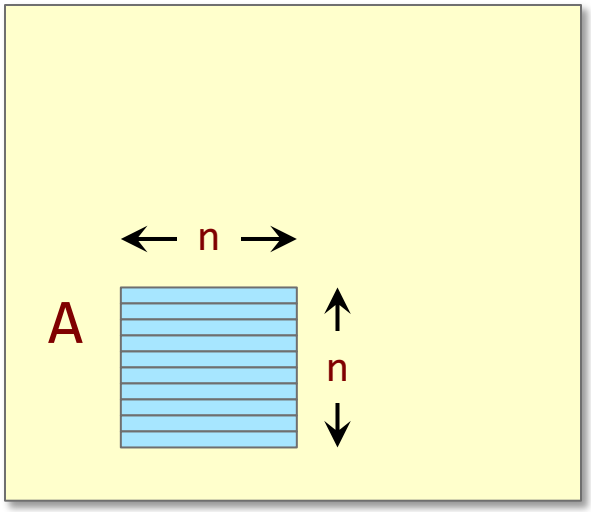
What's Wrong with Short Caches?



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 $n^2 < c\mathcal{M}$!

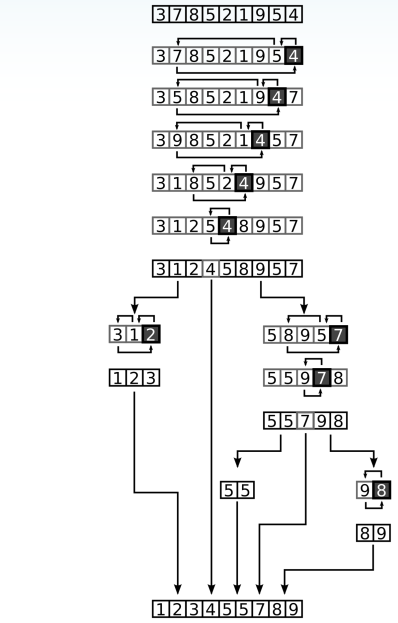
Submatrix Caching Lemma



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

Proof. We have $r = n$, $s_i = n$, $N = n^2$. Since $\mathcal{B}^2 < c\mathcal{M} \leq n^2$, we have $\mathcal{B} \leq n = N/r$. And since $N < \mathcal{M}/3$, the segment caching 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];
}
```

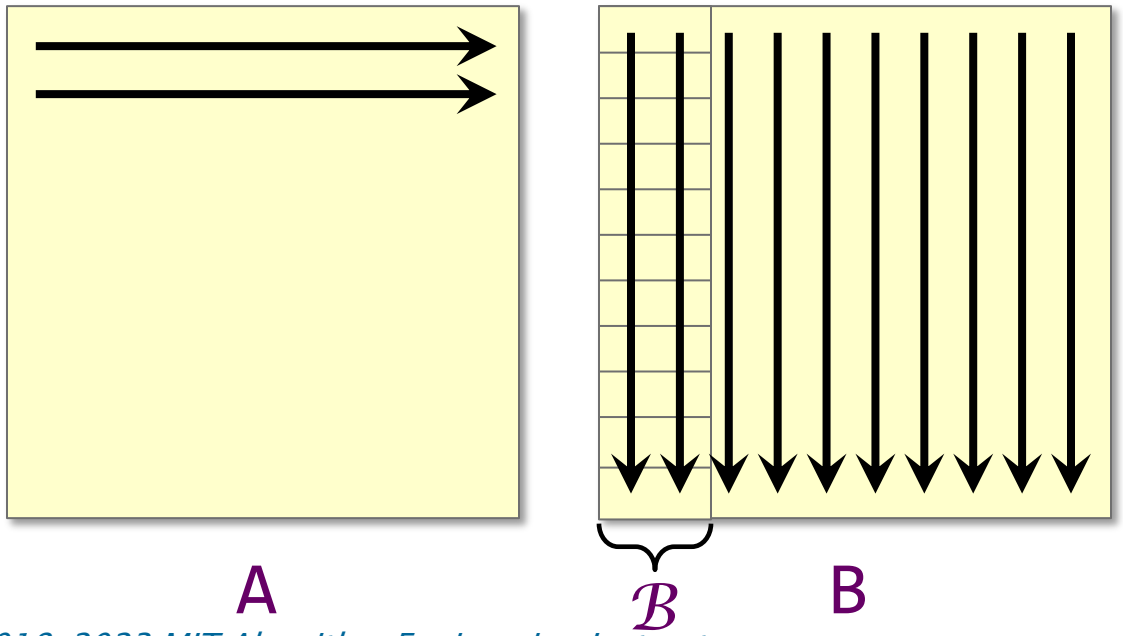
Analysis of work

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

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

$n \geq \mathcal{M}/\mathcal{B}$.

Analyze matrix B.

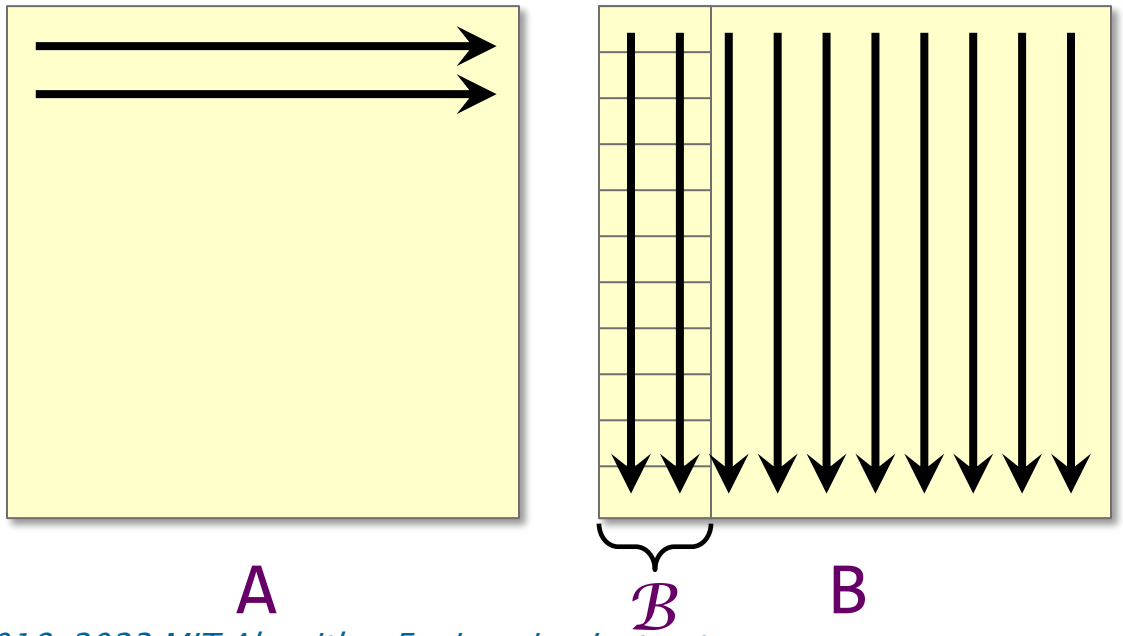
Assume LRU.

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



Case 2

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

Analyze matrix B.

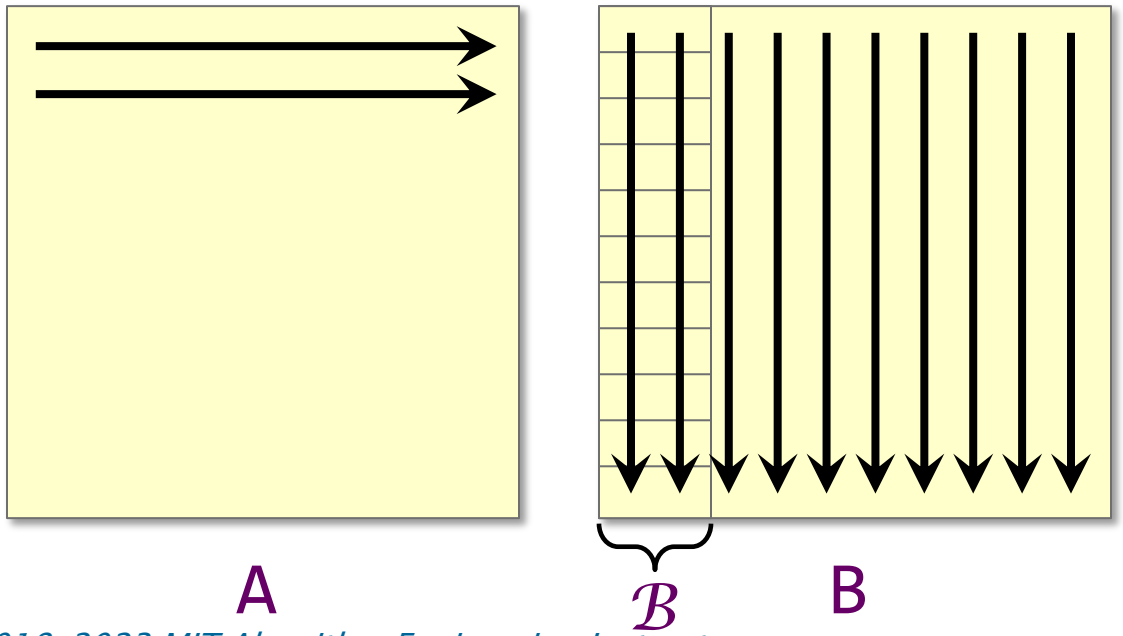
Assume LRU.

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

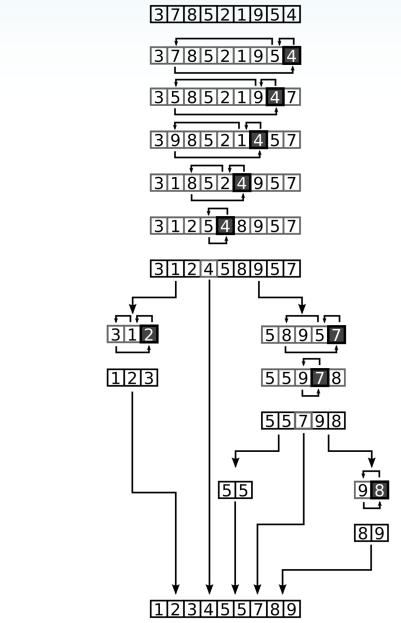
$n < c\mathcal{M}^{1/2}$.

Analyze matrix B.

Assume LRU.

$Q(n) = \Theta(n^2/\mathcal{B})$, by the submatrix caching lemma.

TILING

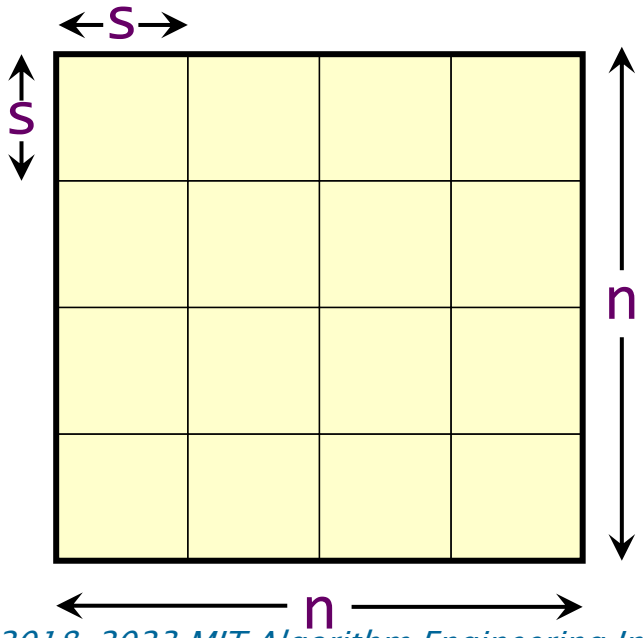


Tiled Matrix Multiplication

```
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];
}
```

outer nest

inner nest



Analysis of work

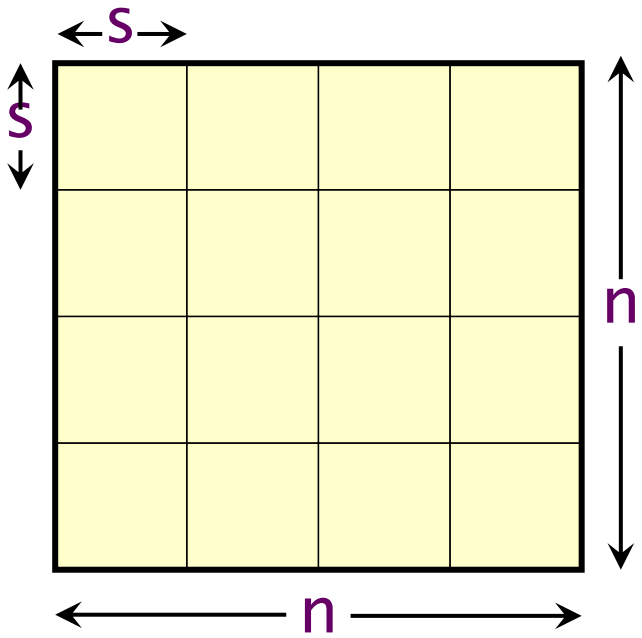
- Work $T(n) = \Theta((n/s)^3(s^3))$
  $= \Theta(n^3)$.

Tiled Matrix Multiplication

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

outer nest

inner nest



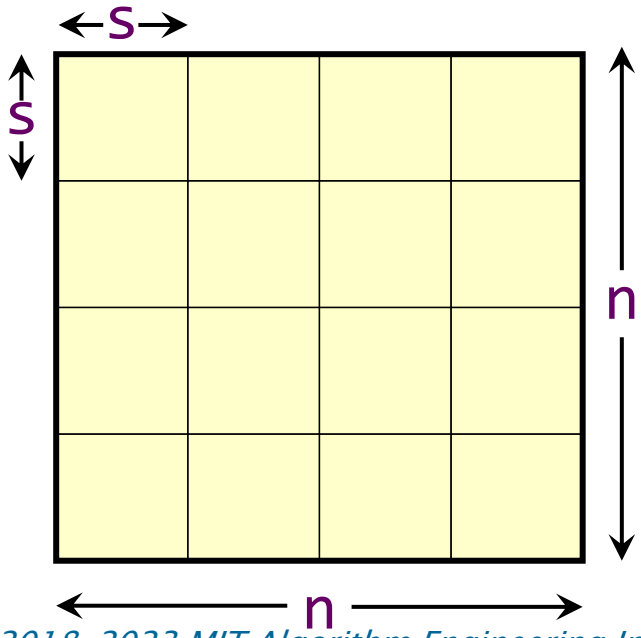
Analysis of cache misses

- Tune s so that the tiles just fit into cache $\Rightarrow s = \Theta(\mathcal{M}^{1/2})$.
- Submatrix caching lemma implies $\Theta(s^2/\mathcal{B})$ misses per tile.
- $Q(n) = \Theta((n/s)^3(s^2/\mathcal{B}))$
  $= \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$.
- Optimal [HK81].

Tiled Matrix Multiplication

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

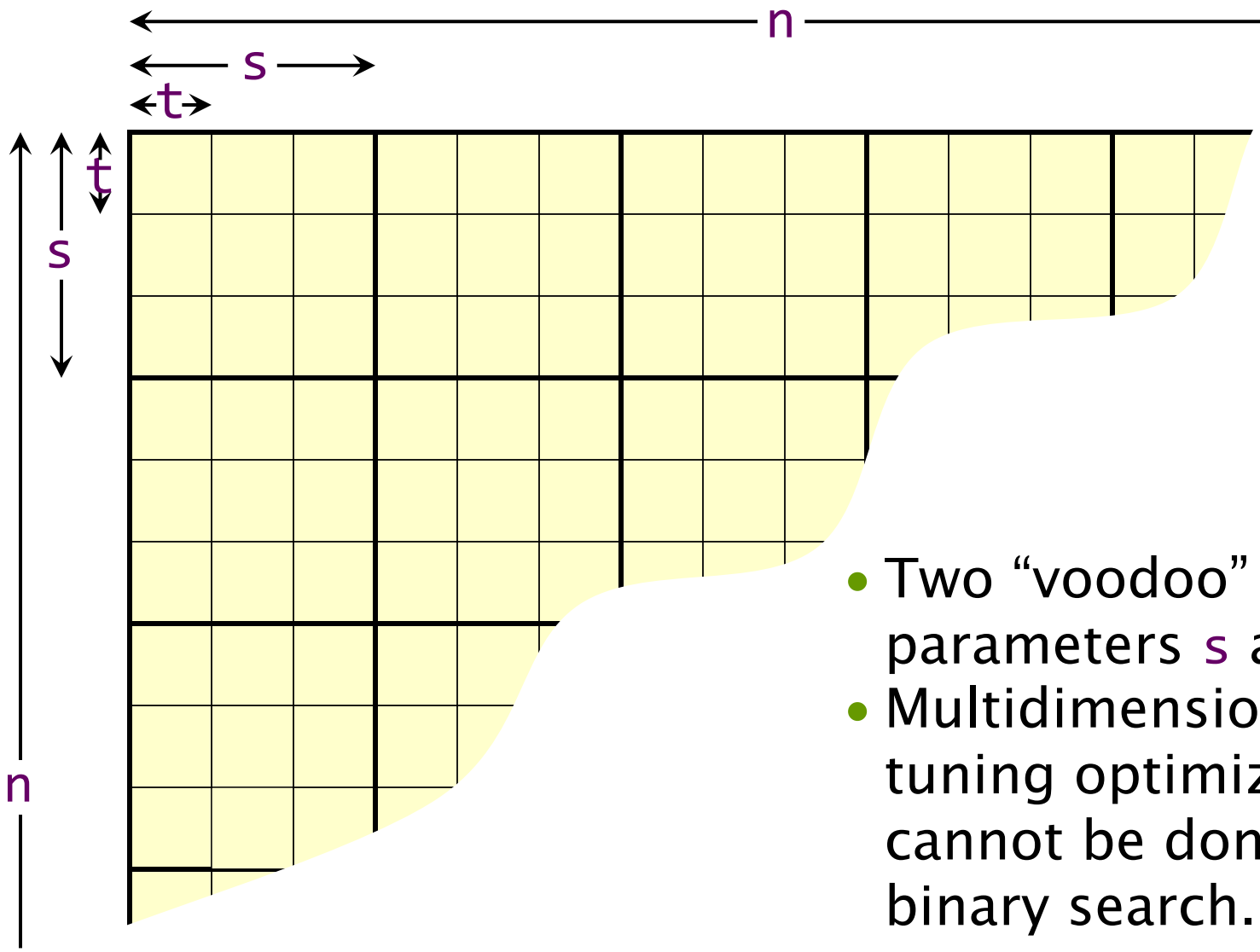
Voodoo!



Analysis of cache misses

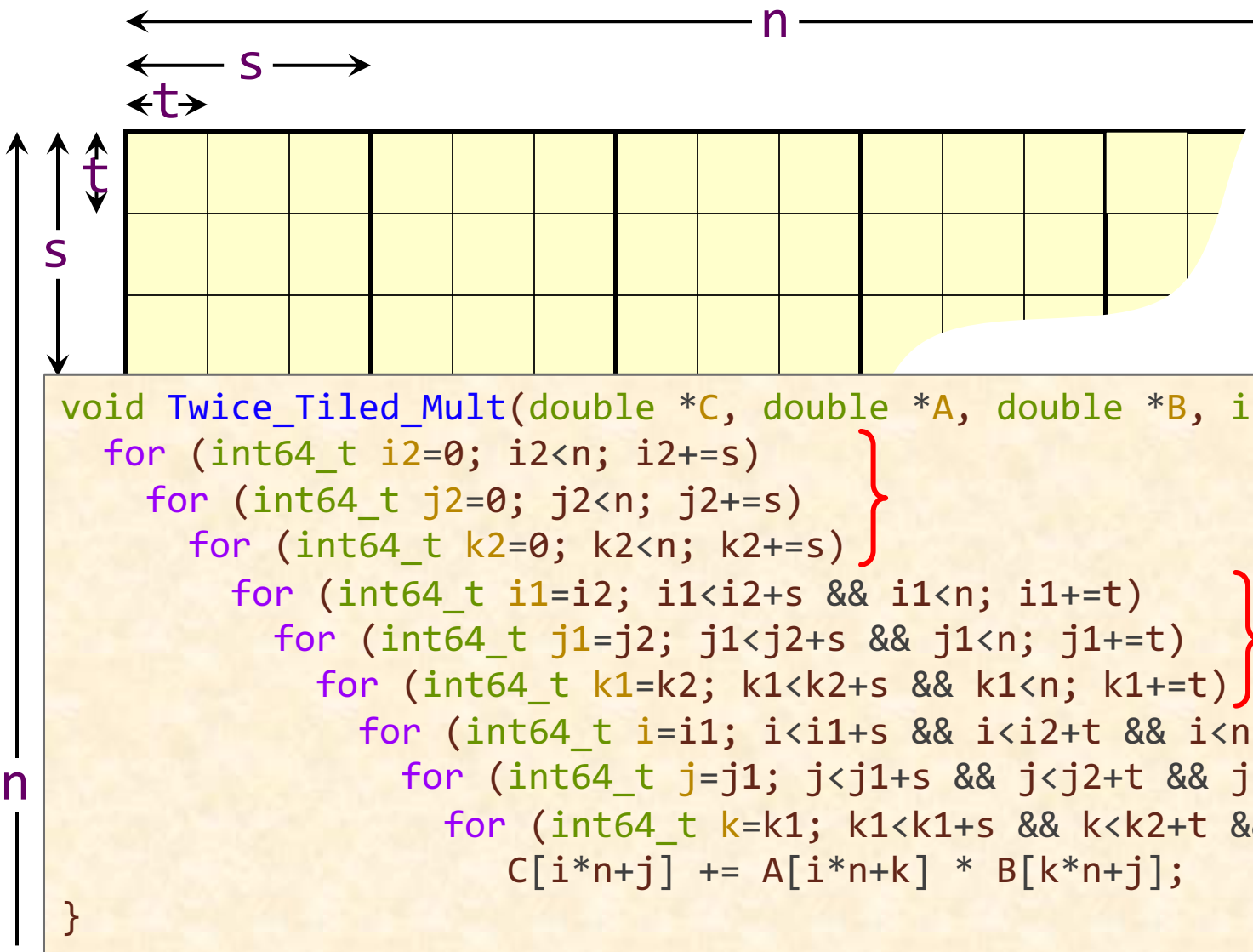
- Tune s so that the tiles just fit into cache $\Rightarrow s = \Theta(\mathcal{M}^{1/2})$.
- Submatrix caching lemma implies $\Theta(s^2/\mathcal{B})$ misses per tile.
- $Q(n) = \Theta((n/s)^3(s^2/\mathcal{B}))$
  $= \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$.
- Optimal [HK81].

Two-Level Cache



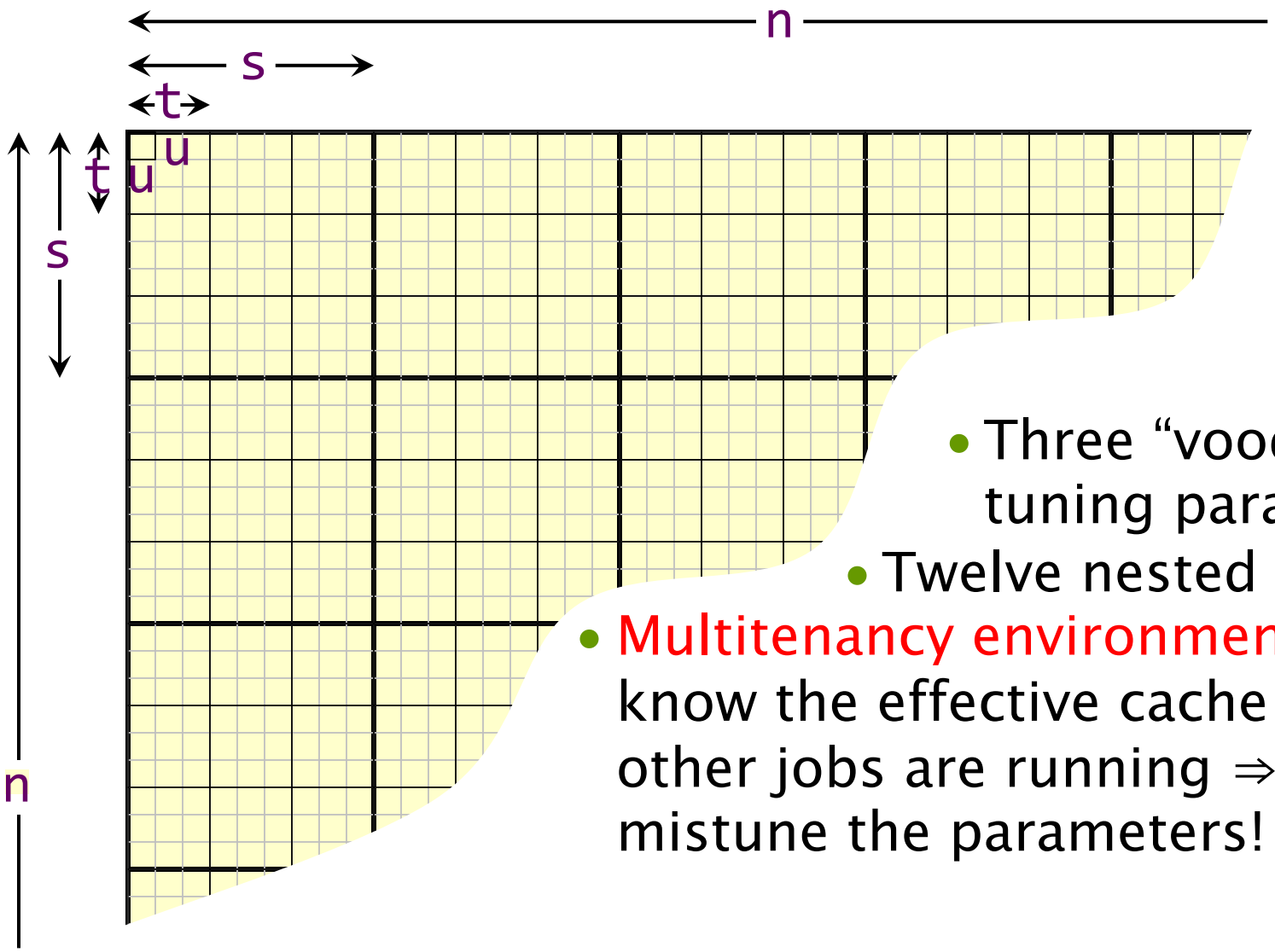
- Two “voodoo” tuning parameters `s` and `t`.
- Multidimensional tuning optimization cannot be done with binary search.

Two-Level Cache



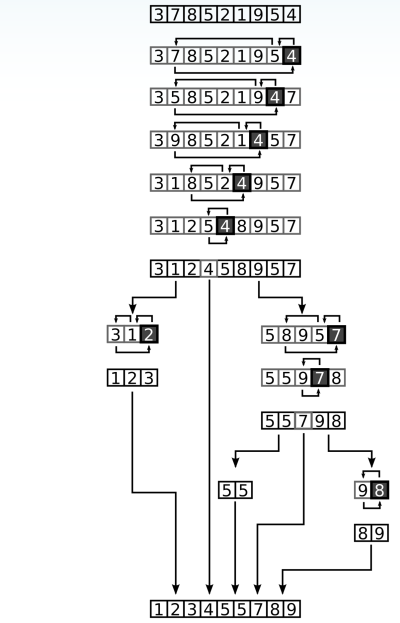

```
void Twice_Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i2=0; i2<n; i2+=s)
    for (int64_t j2=0; j2<n; j2+=s)
      for (int64_t k2=0; k2<n; k2+=s)
        for (int64_t i1=i2; i1<i2+s && i1<n; i1+=t)
          for (int64_t j1=j2; j1<j2+s && j1<n; j1+=t)
            for (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.
- Multitenancy 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.

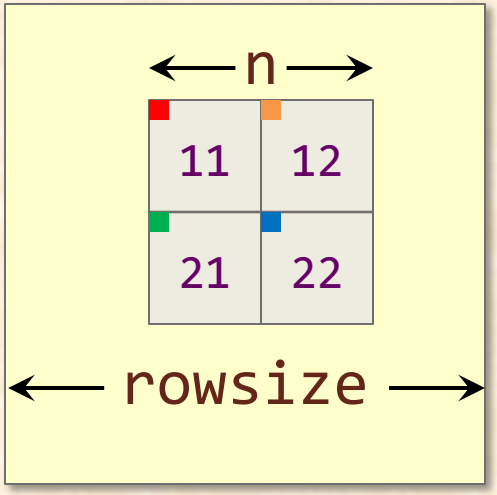
Recursive Code

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

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

Analysis of Work

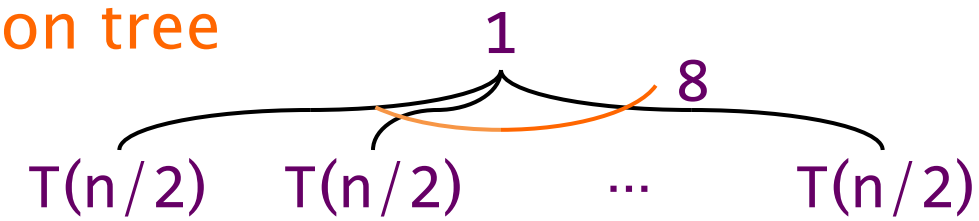
$$T(n) = 8T(n/2) + 1$$

recursion tree $T(n)$

Analysis of Work

$$T(n) = 8T(n/2) + 1$$

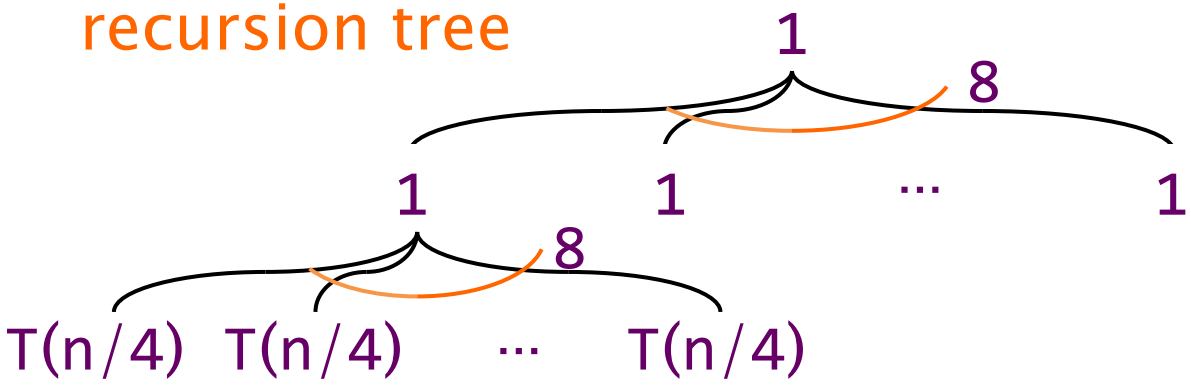
recursion tree



Analysis of Work

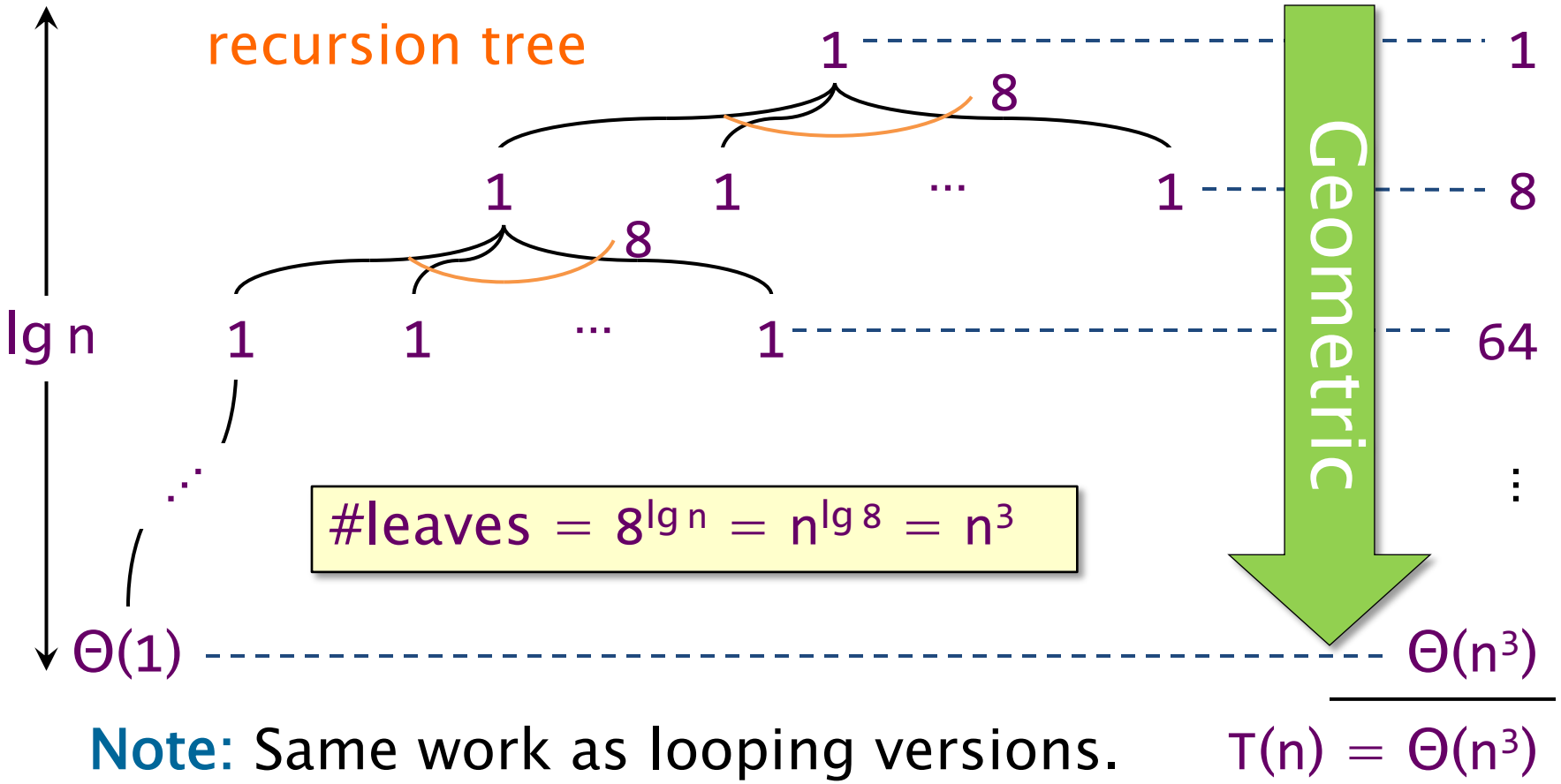
$$T(n) = 8T(n/2) + 1$$

recursion tree



Analysis of Work

$$T(n) = 8T(n/2) + 1$$



Note: Same work as looping versions.

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

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

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

Analysis of Cache Misses

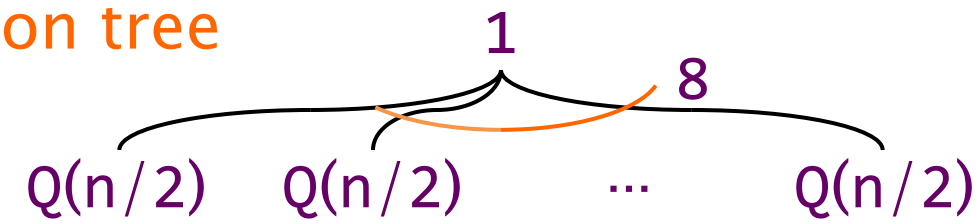
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + 1 \text{ otherwise.} \end{cases}$$

recursion tree $Q(n)$

Analysis of Cache Misses

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

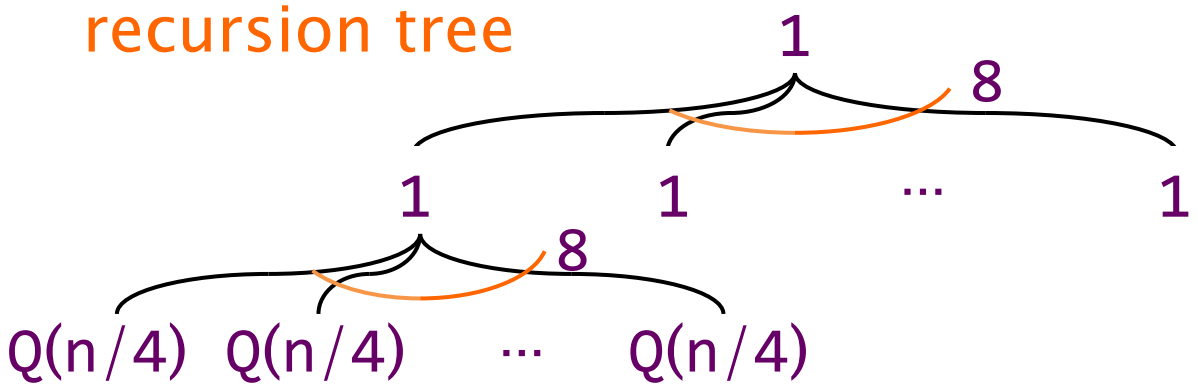
recursion tree



Analysis of Cache Misses

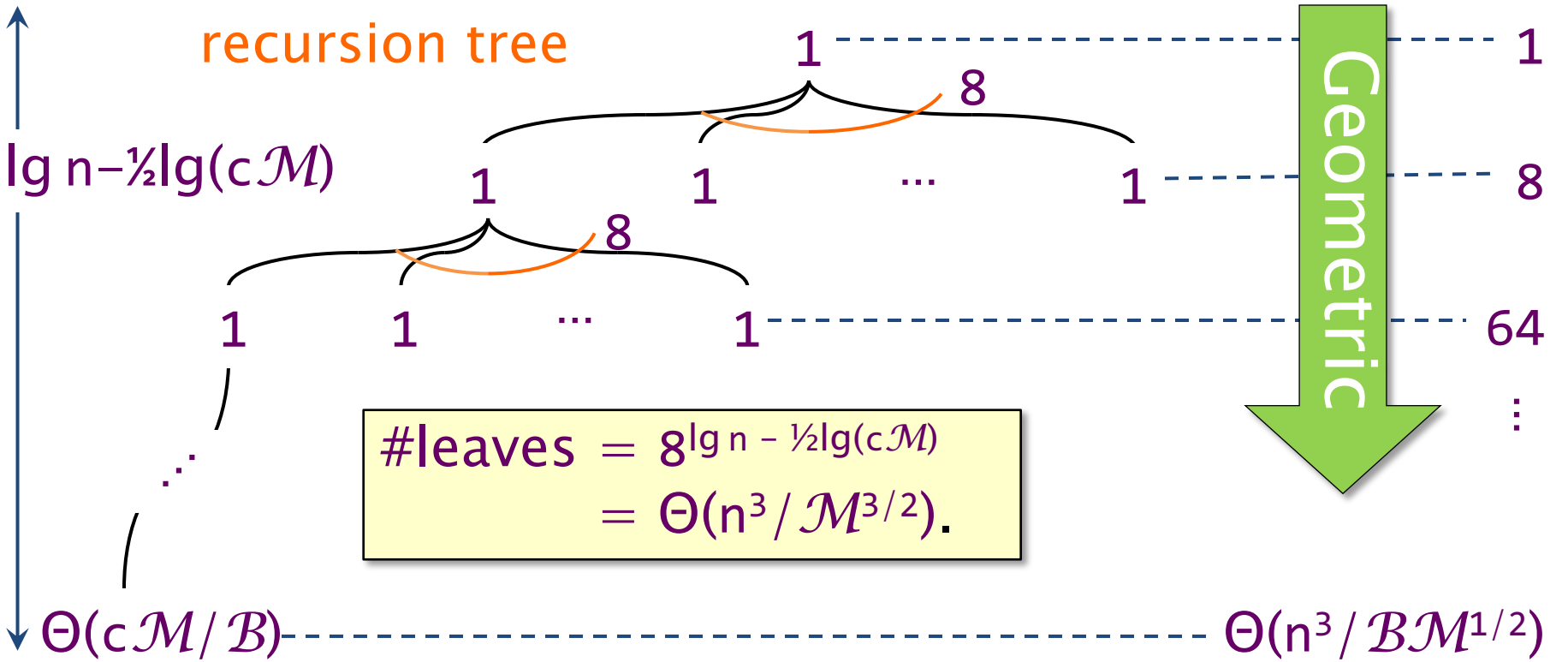
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8\,Q(n/2) + 1 \text{ otherwise.} \end{cases}$$

recursion tree



Analysis of Cache Misses

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



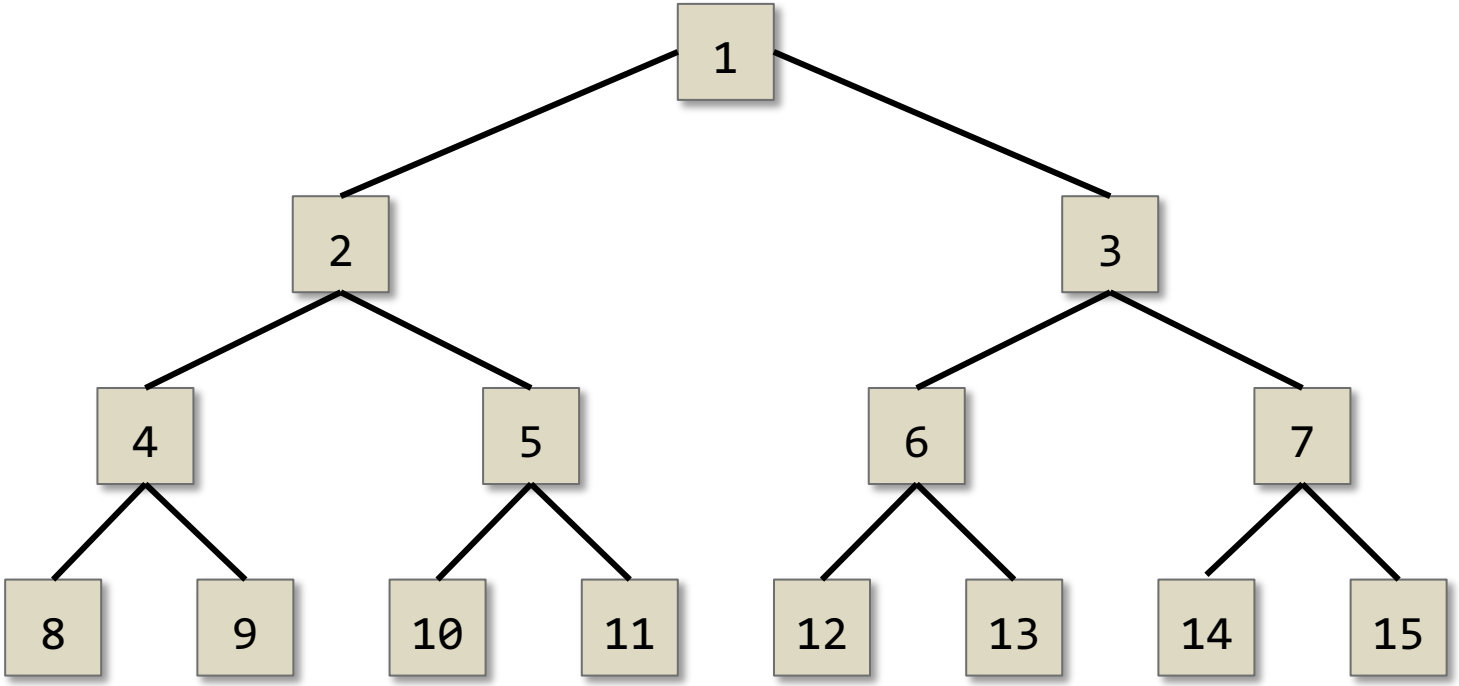
Same cache misses as with tiling!

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

Cache-Oblivious Algorithms

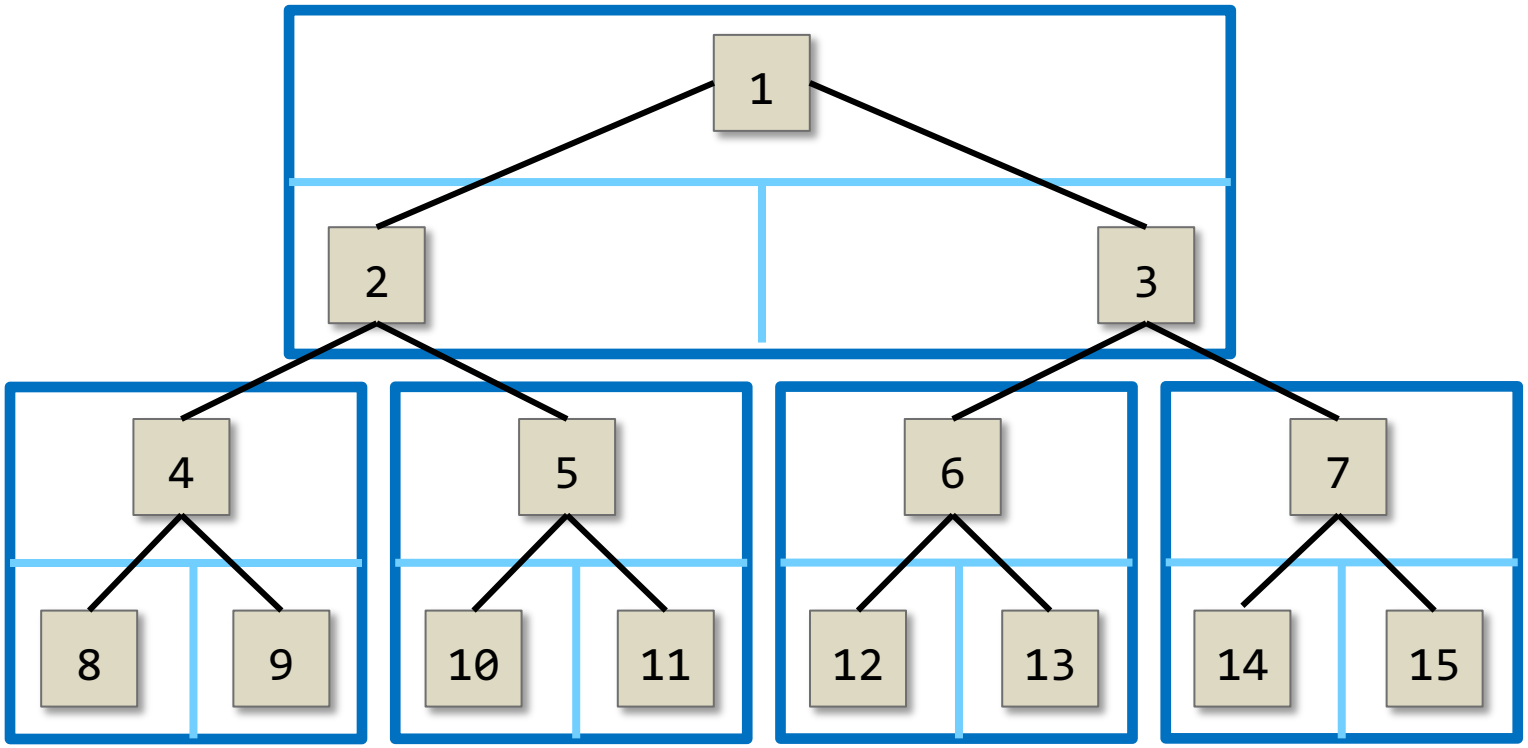
- **Cache-oblivious algorithms** [FLPR99]
- No voodoo tuning parameters.
- No explicit knowledge of caches.
- Passively autotune.
- Handle multilevel caches automatically.
- Good in multitenancy environments.

Cache-Aware Search Tree (Static)



Cache misses: $Q(n) = \Theta(\lg n)$

Cache-Oblivious Search Tree (Static)



1	2	3	4	8	9	5	10	12	6	12	13	7	14	15

Cache misses: $Q(n) = \Theta(\log_{\mathcal{B}} n)$

Other C–O Algorithms

Matrix Transposition/Addition $\Theta(1+mn/\mathcal{B})$

Straightforward recursive algorithm.

Strassen's Algorithm $\Theta(n + n^2/\mathcal{B} + n^{\lg 7}/\mathcal{B}\mathcal{M}^{(\lg 7)/2 - 1})$

Straightforward recursive algorithm.

Fast Fourier Transform $\Theta(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$

Variant of Cooley–Tukey [CT65] using cache-oblivious matrix transpose.

LUP-Decomposition $\Theta(1 + n^2/\mathcal{B} + n^3/\mathcal{B}\mathcal{M}^{1/2})$

Recursive algorithm due to Sivan Toledo [T97].

C-O Data Structures

Ordered-File Maintenance

$O(1 + (\lg^2 n)/\mathcal{B})$

INSERT/DELETE anywhere in file while maintaining $O(1)$-sized gaps. Amortized bound [BDFC00], later improved in [BCDFC02].

B-Trees

INSERT/DELETE:	$O(1+\log_{\mathcal{B}+1} n+(\lg^2 n)/\mathcal{B}$
SEARCH:	$O(1+\log_{\mathcal{B}+1} n)$
TRAVERSE:	$O(1+k/\mathcal{B})$

Solution [BDFC00] with later simplifications [BDIW02], [BFJ02].

Priority Queues

$O(1+(1/\mathcal{B})\log_{\mathcal{M}/\mathcal{B}}(n/\mathcal{B}))$

Funnel-based solution [BF02]. General scheme based on buffer trees [ABDHMM02] supports INSERT/DELETE.

Other C–O Algorithms

Matrix Transposition/Addition $\Theta(1+mn/\mathcal{B})$

Straightforward recursive algorithm.

Strassen's Algorithm $\Theta(n + n^2/\mathcal{B} + n^{\lg 7}/\mathcal{B}\mathcal{M}^{(\lg 7)/2 - 1})$

Straightforward recursive algorithm.

Fast Fourier Transform $\Theta(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$

Variant of Cooley–Tukey [CT65] using cache-oblivious matrix transpose.

LUP-Decomposition $\Theta(1 + n^2/\mathcal{B} + n^3/\mathcal{B}\mathcal{M}^{1/2})$

Recursive algorithm due to Sivan Toledo [T97].

C-O Data Structures

Ordered-File Maintenance

$O(1 + (\lg^2 n)/\mathcal{B})$

INSERT/DELETE anywhere in file while maintaining $O(1)$-sized gaps. Amortized bound [BDFC00], later improved in [BCDFC02].

B-Trees

INSERT/DELETE:	$O(1+\log_{\mathcal{B}+1} n+(\lg^2 n)/\mathcal{B}$
SEARCH:	$O(1+\log_{\mathcal{B}+1} n)$
TRAVERSE:	$O(1+k/\mathcal{B})$

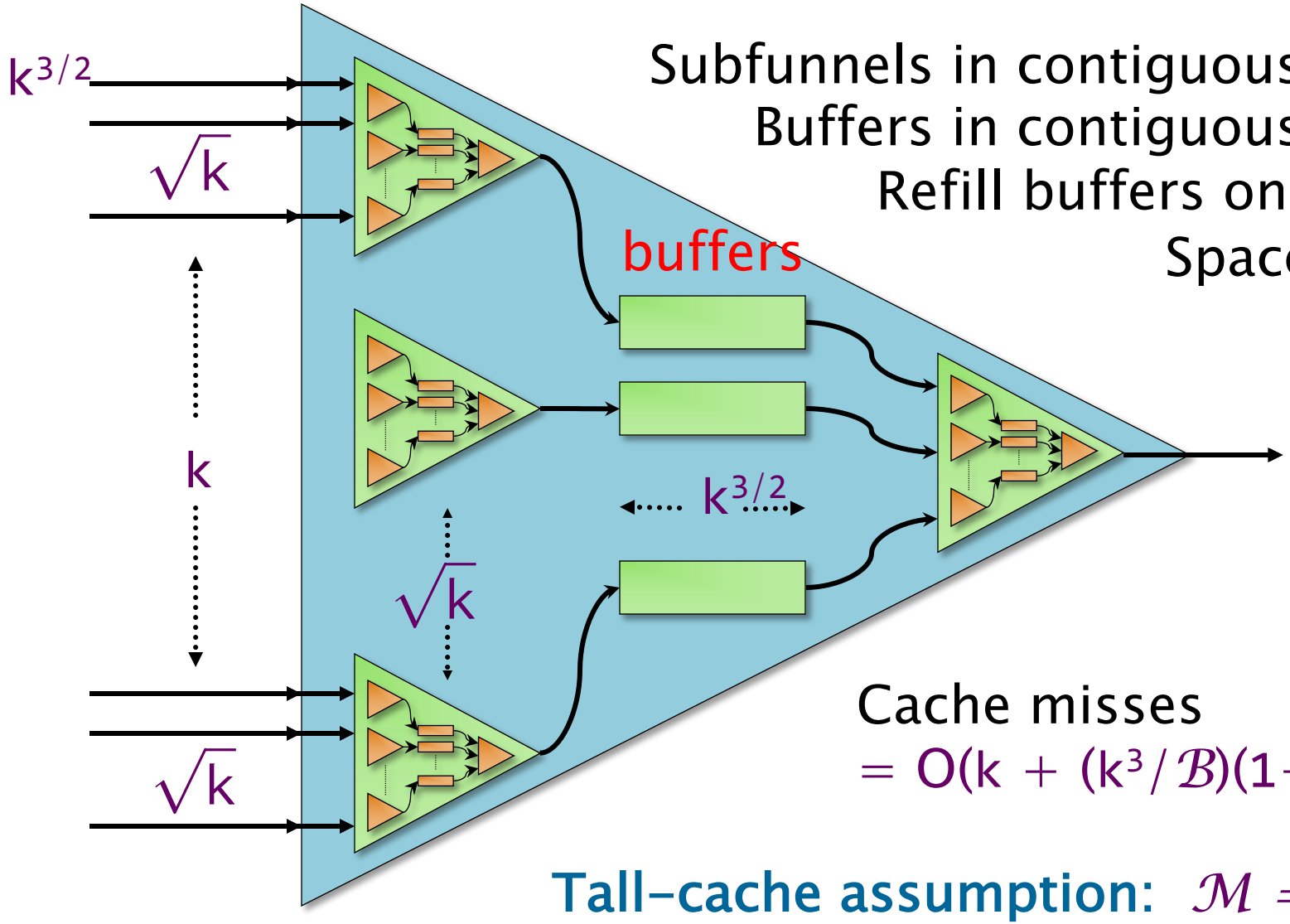
Solution [BDFC00] with later simplifications [BDIW02], [BFJ02].

Priority Queues

$O(1+(1/\mathcal{B})\log_{\mathcal{M}/\mathcal{B}}(n/\mathcal{B}))$

Funnel-based solution [BF02]. General scheme based on buffer trees [ABDHMM02] supports INSERT/DELETE.

Construction of a k-funnel



Subfunnels in contiguous storage.
Buffers in contiguous storage.
Refill buffers on demand.
Space $= O(k^2)$.

Cache misses
$= O(k + (k^3/\mathcal{B})(1+\log_{\mathcal{M}} k))$.

Tall-cache assumption: $\mathcal{M} = \Omega(\mathcal{B}^2)$.