6.172
Performance Engineering of Software Systems

LECTURE 8
Cache-Efficient Algorithms

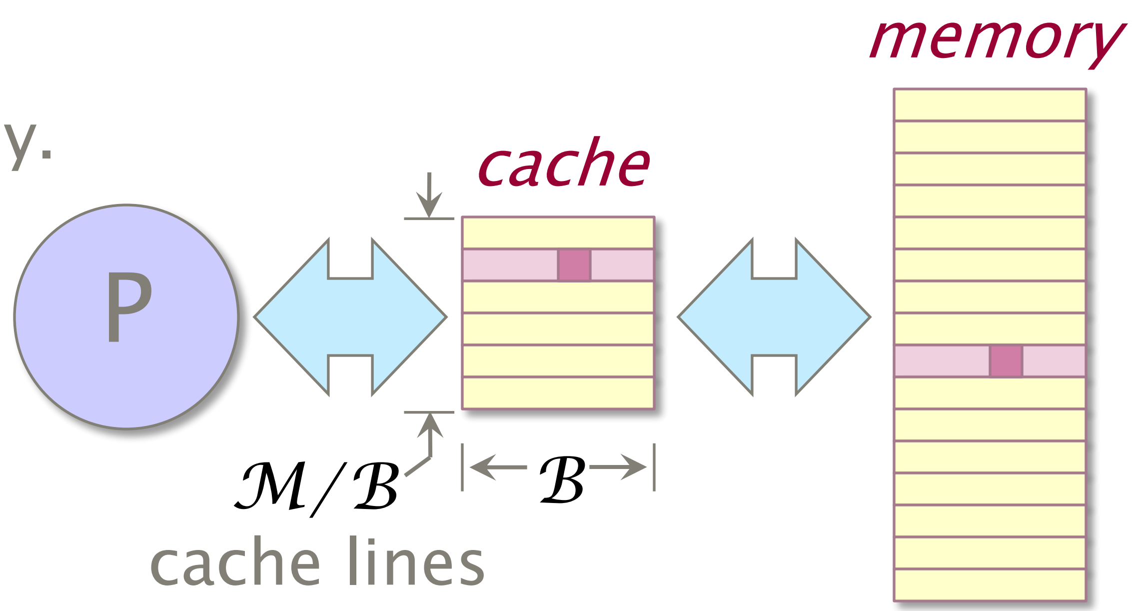
Charles E. Leiserson

October 5, 2010

Ideal-Cache Model

Recall:

- 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. ■

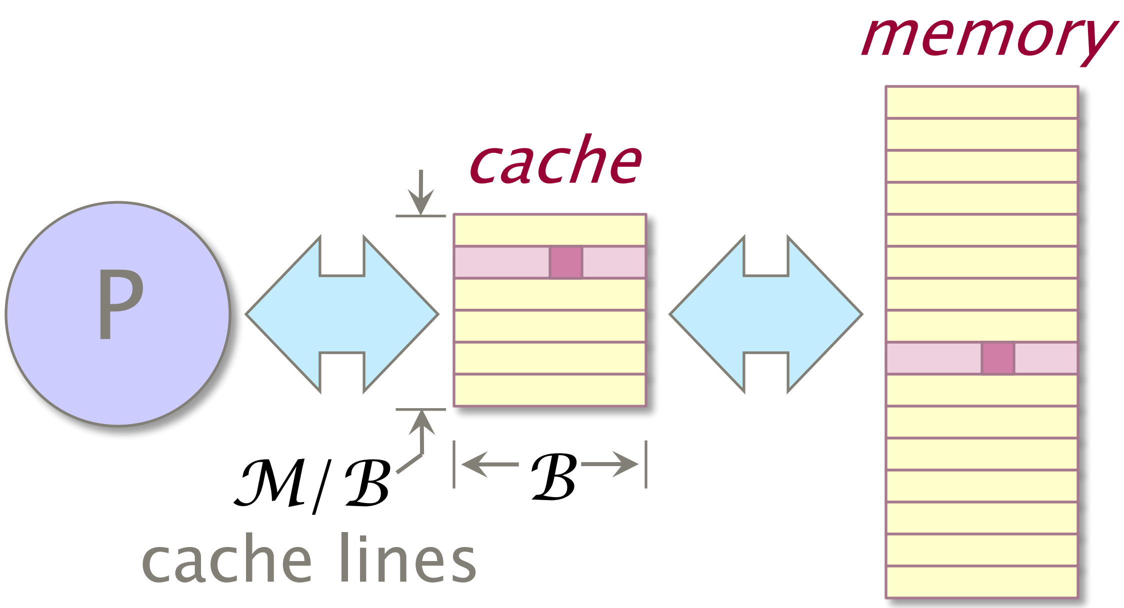
Implication

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.

Tall Caches



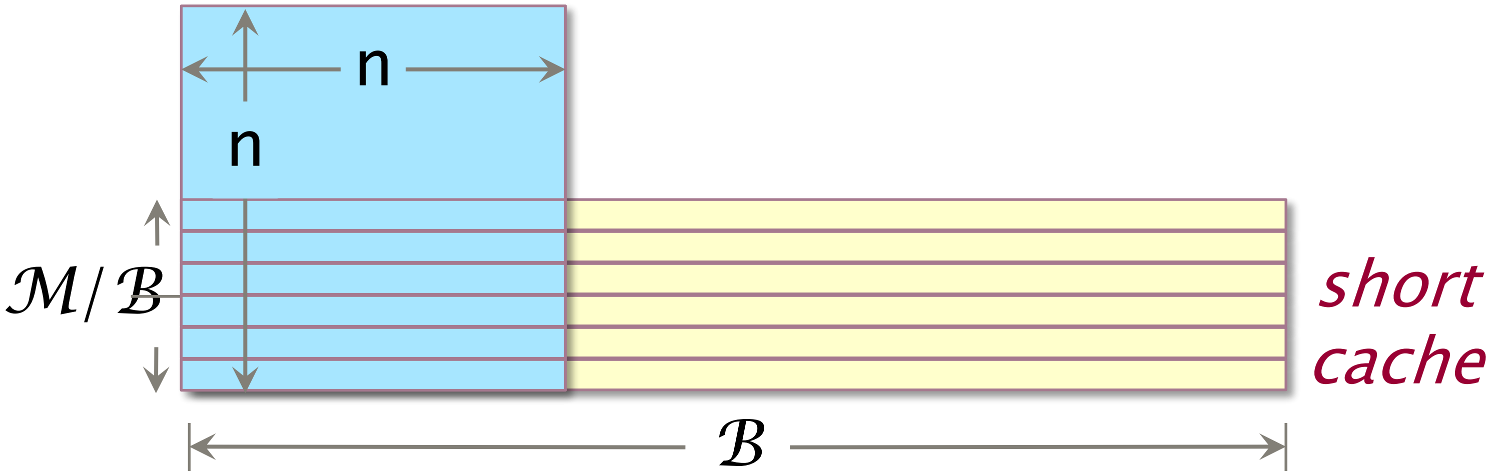
Tall-cache assumption

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

Example: Intel Core i7 (Nehalem)

- 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 $c \le 1$.

An $n \times n$ matrix stored in row-major order may not fit in a short cache even if $n^2 < c\mathcal{M}$! Such a matrix always fits in a tall cache, and if $n = \Omega(\mathcal{B})$, it takes at most $\Theta(n^2/\mathcal{B})$ cache misses to load it in.

Multiply n×n Matrices

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

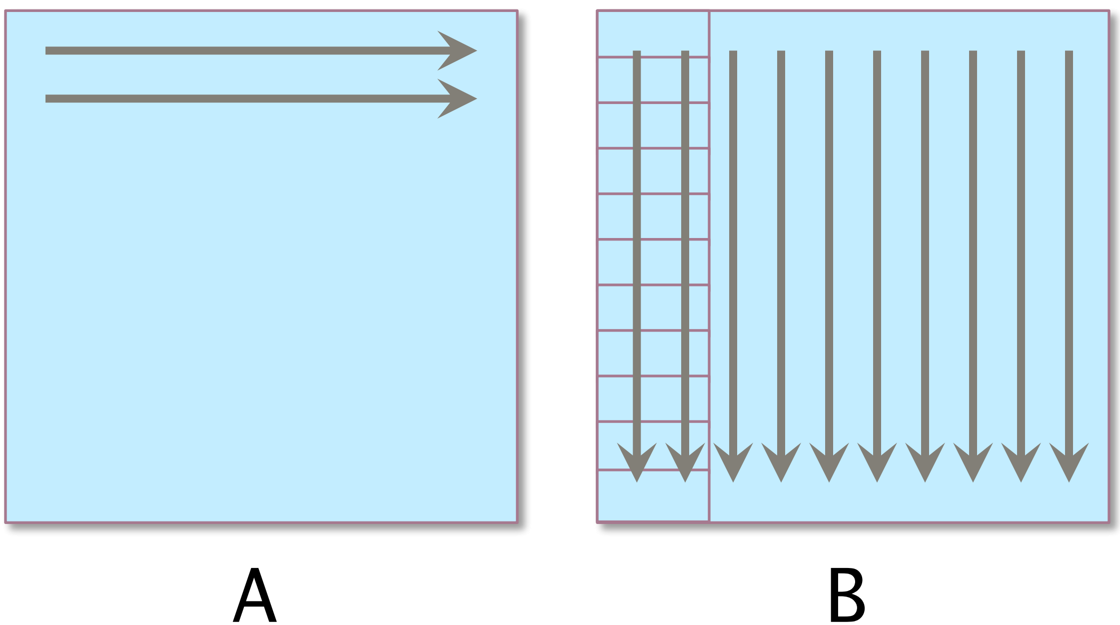
Analysis of work

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

Analysis of Cache Misses

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

row-major layout of arrays



Case 1:
n > $\mathcal{M}/\mathcal{B}$.

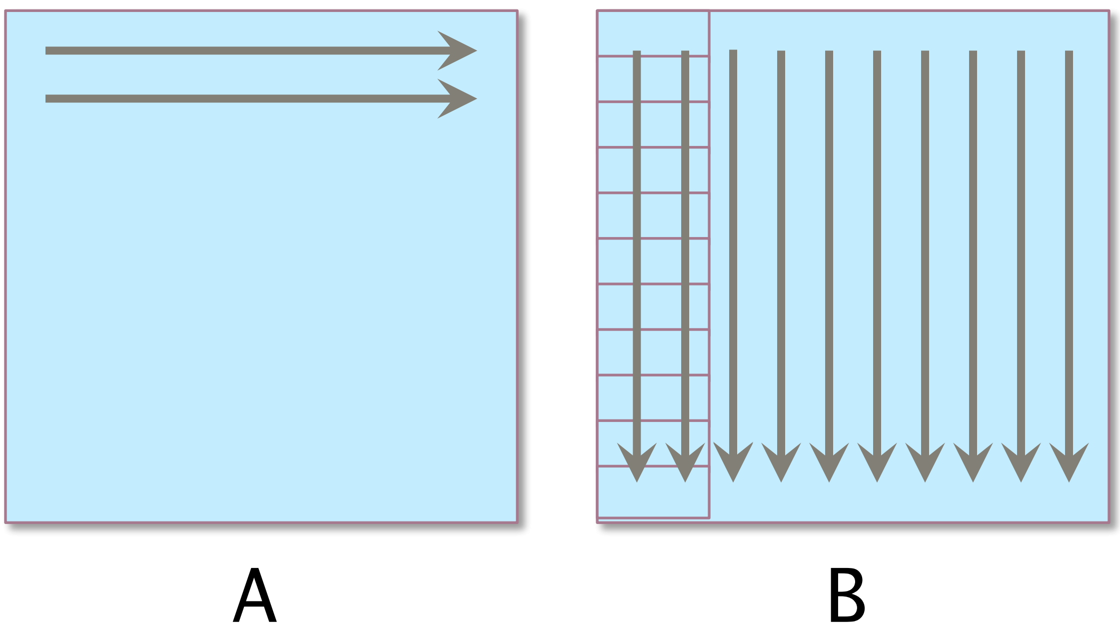
Assume LRU.

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

Analysis of Cache Misses

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

row-major layout of arrays



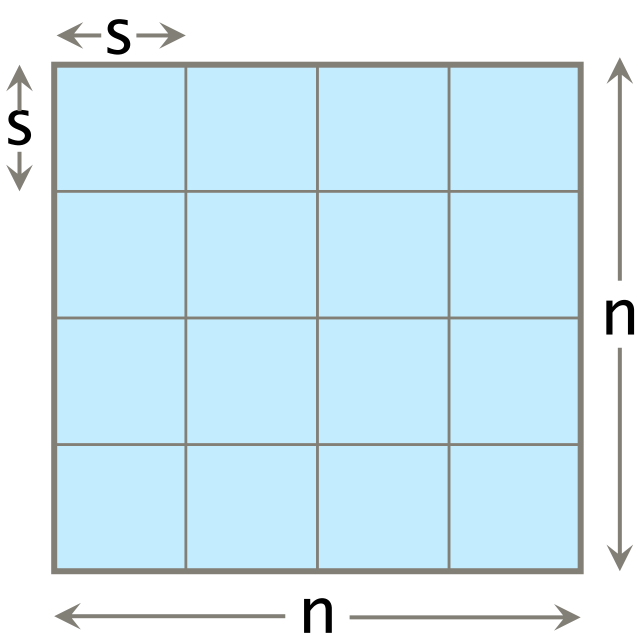
Case 2:
$\mathcal{M}^{1/2} < n < c\mathcal{M}/\mathcal{B}$.

Assume LRU.

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

Tiled Matrix Multiplication

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



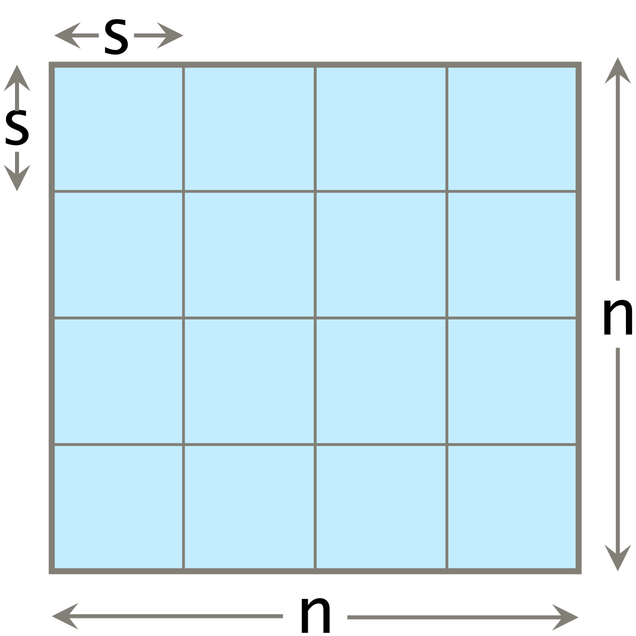
Analysis of work

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

  $= \Theta(n^3)$.

Tiled Matrix Multiplication

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



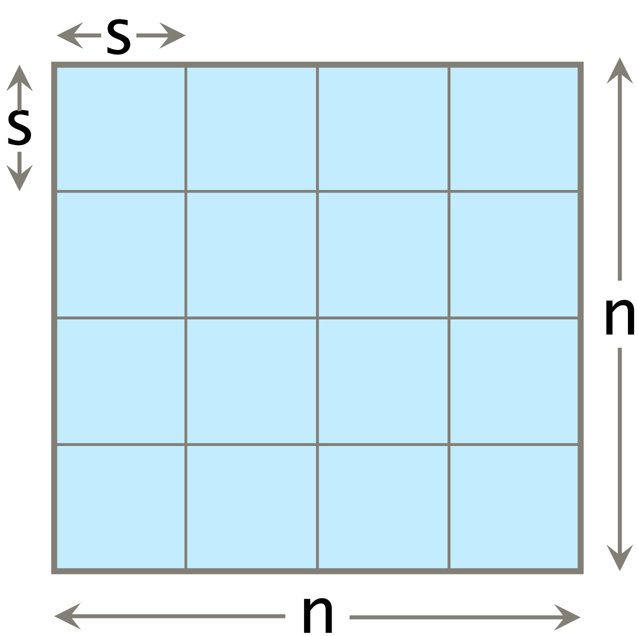
Analysis of cache misses

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

Tiled Matrix Multiplication

```
void Tiled_Mult(double *C, double *A, double *B, int n) {
  for (int i1
    for (
      for

          fo
            C[i*n+j]                        n+j];
}
```

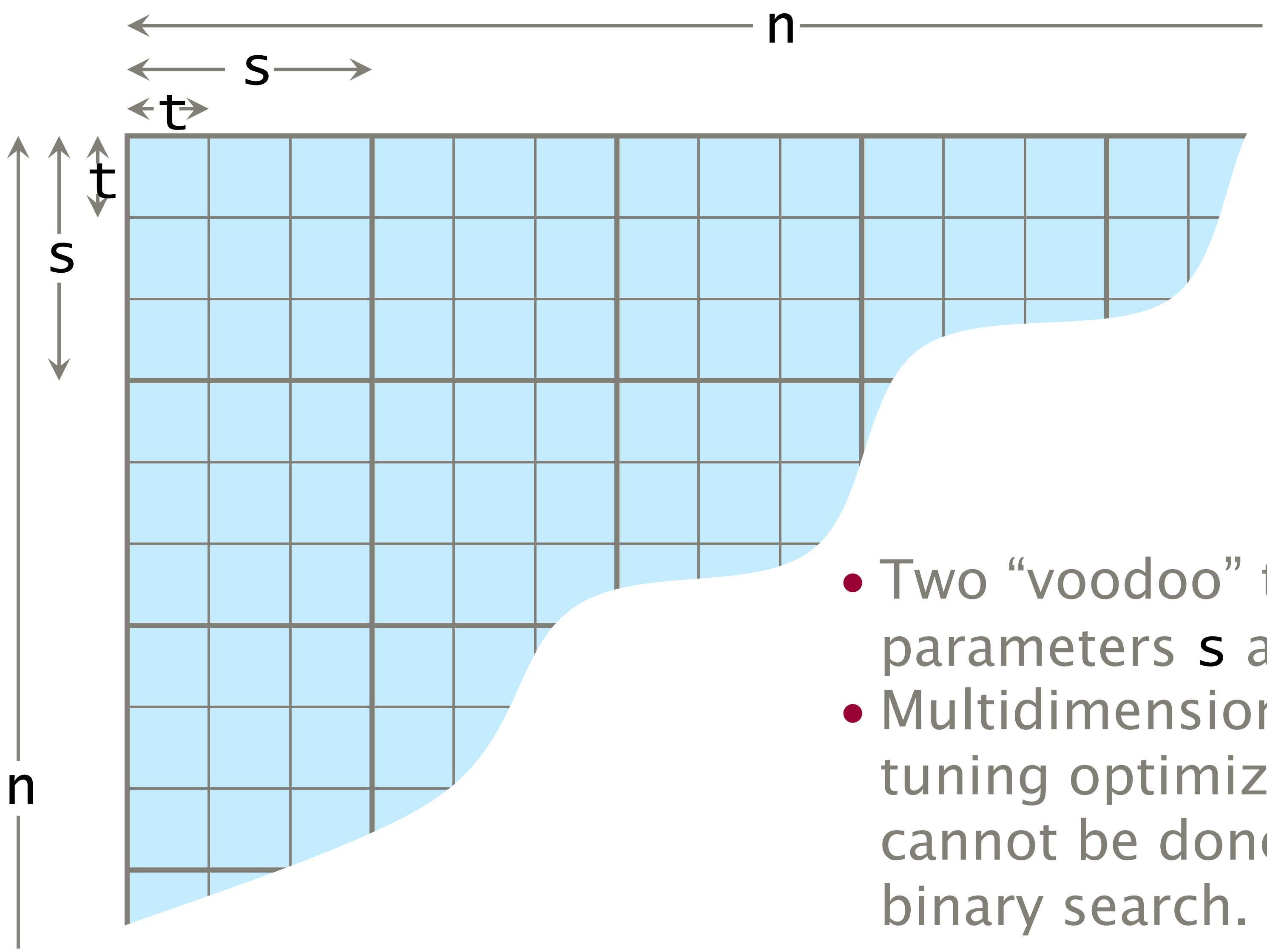
Voodoo!



Analysis of cache misses

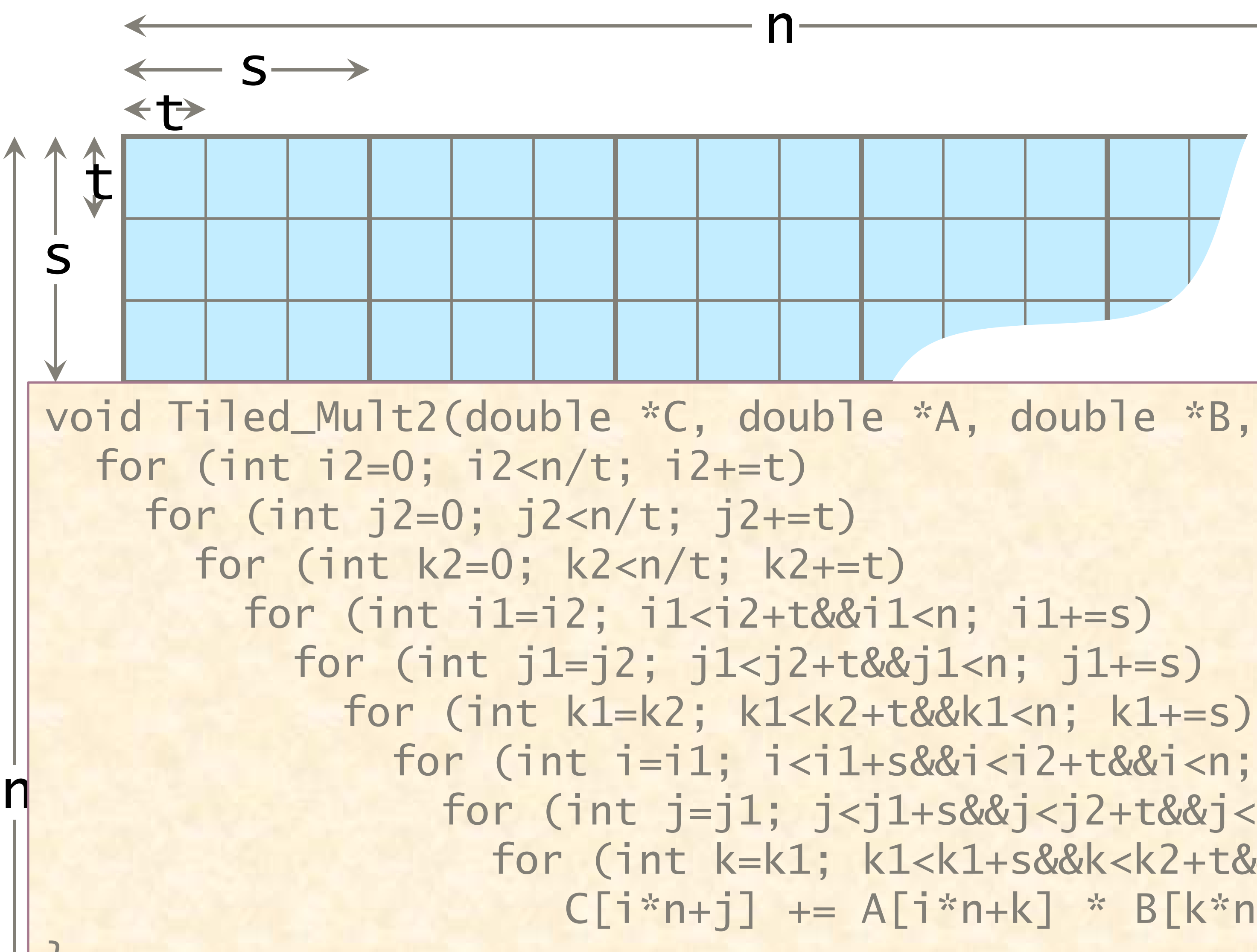
- Tune s so that the submatrices just fit into cache ⇒ $s = \Theta(\mathcal{M}^{1/2})$.
- Tall-cache assumption implies $\Theta(s^2/\mathcal{B})$ misses per submatrix.
- $Q(n) = \Theta((n/s)^3(s^2/\mathcal{B}))$
  $= \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$. *Remember this!*
- Optimal [HK81].

Two-Level Cache



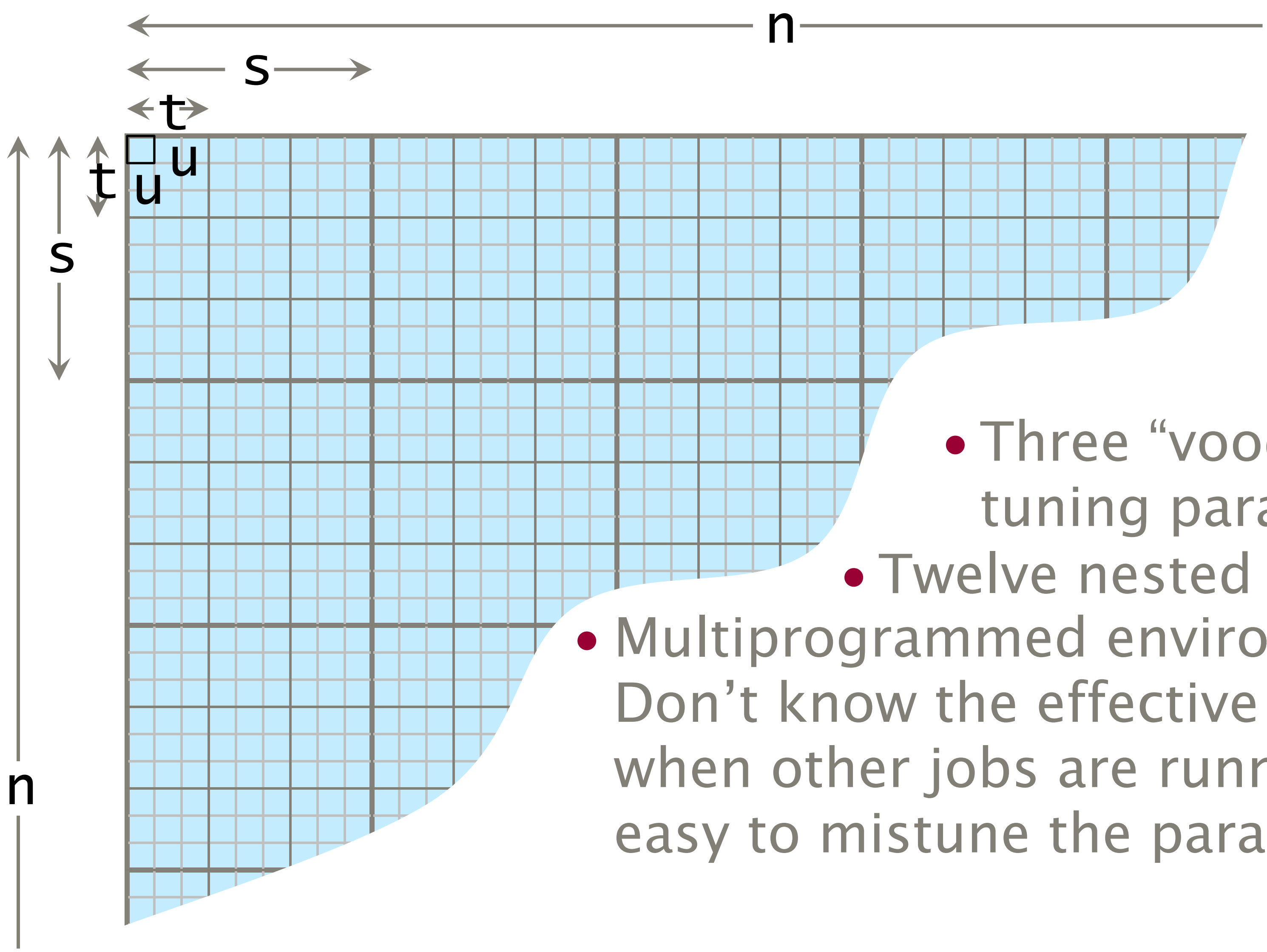
- Two "voodoo" tuning parameters `s` and `t`.
- Multidimensional tuning optimization cannot be done with binary search.

Two-Level Cache




```
void Tiled_Mult2(double *C, double *A, double *B, int n) {
  for (int i2=0; i2<n/t; i2+=t)
    for (int j2=0; j2<n/t; j2+=t)
      for (int k2=0; k2<n/t; k2+=t)
        for (int i1=i2; i1<i2+t&&i1<n; i1+=s)
          for (int j1=j2; j1<j2+t&&j1<n; j1+=s)
            for (int k1=k2; k1<k2+t&&k1<n; k1+=s)
              for (int i=i1; i<i1+s&&i<i2+t&&i<n; i++)
                for (int j=j1; j<j1+s&&j<j2+t&&j<n; j++)
                  for (int 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!

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.

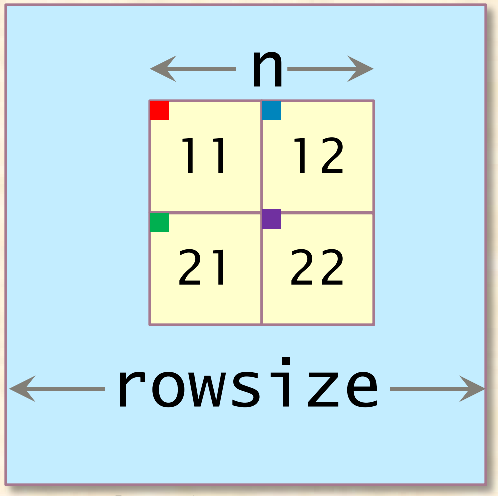
Recursive Code

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

$$W(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8W(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

Analysis of Work

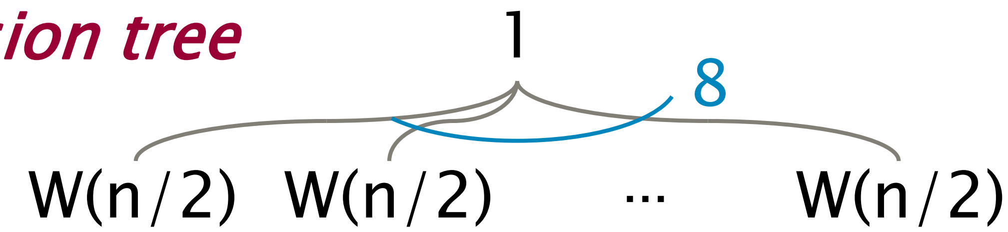
$$W(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8W(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

recursion tree $W(n)$

Analysis of Work

$$W(n) = \begin{cases} \Theta(1) \text{ if } n = 1, \\ 8W(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

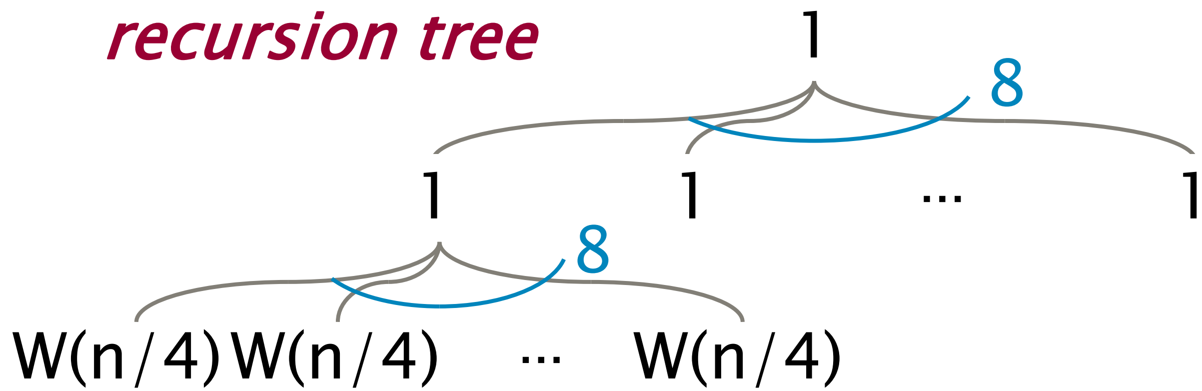
recursion tree



Analysis of Work

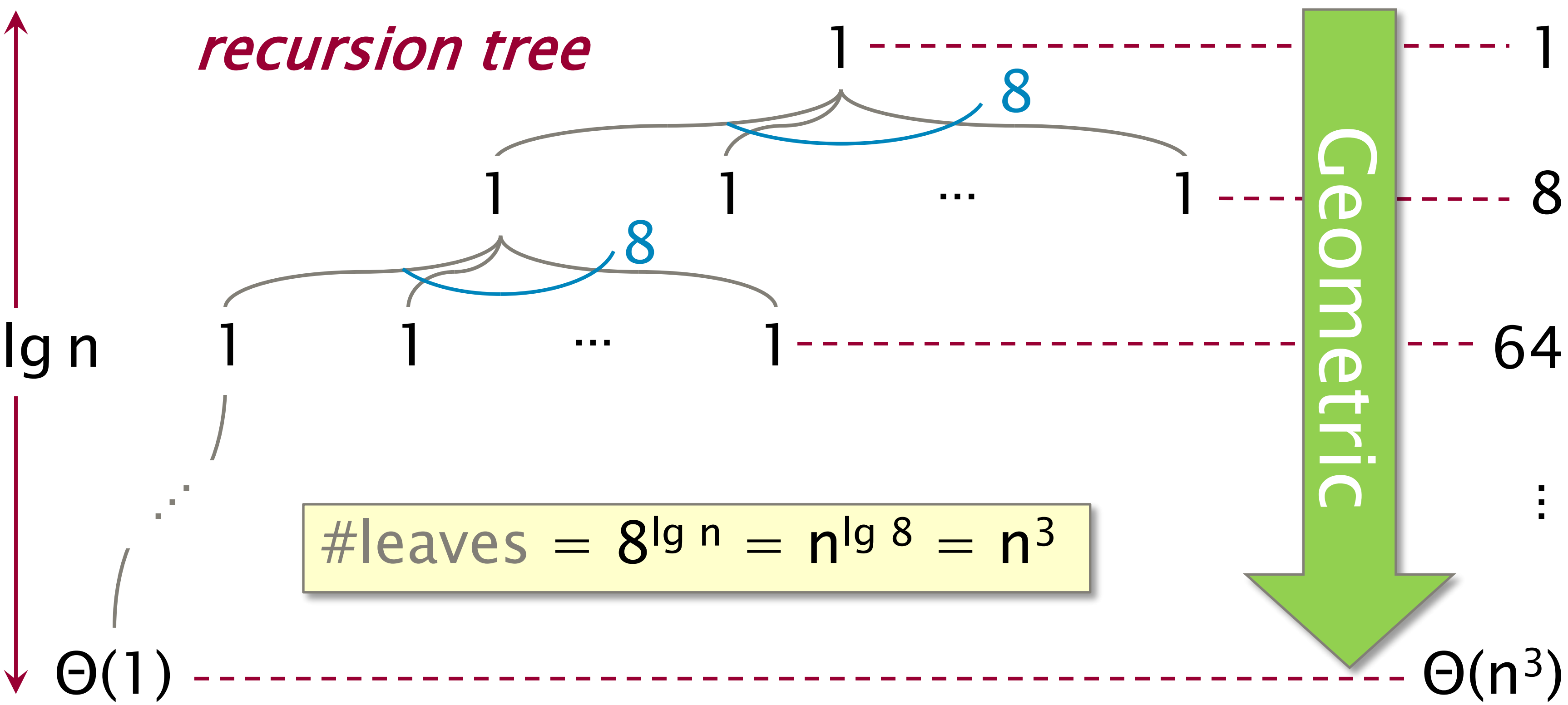
$$W(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8W(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

recursion tree



Analysis of Work

$$W(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8W(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$



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

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

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,
              int n, int rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int d11 = 0;
    int d12 = n/2;
    int d21 = (n/2) * rowsize;
    int 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);
} }
```

Tall-cache assumption

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

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

1

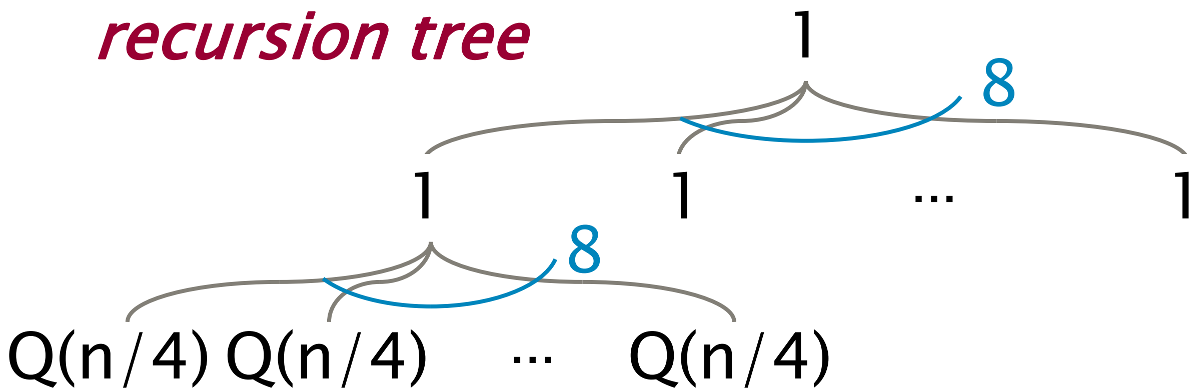
8

$Q(n/2)$ $Q(n/2)$ … $Q(n/2)$

Analysis of Cache Misses

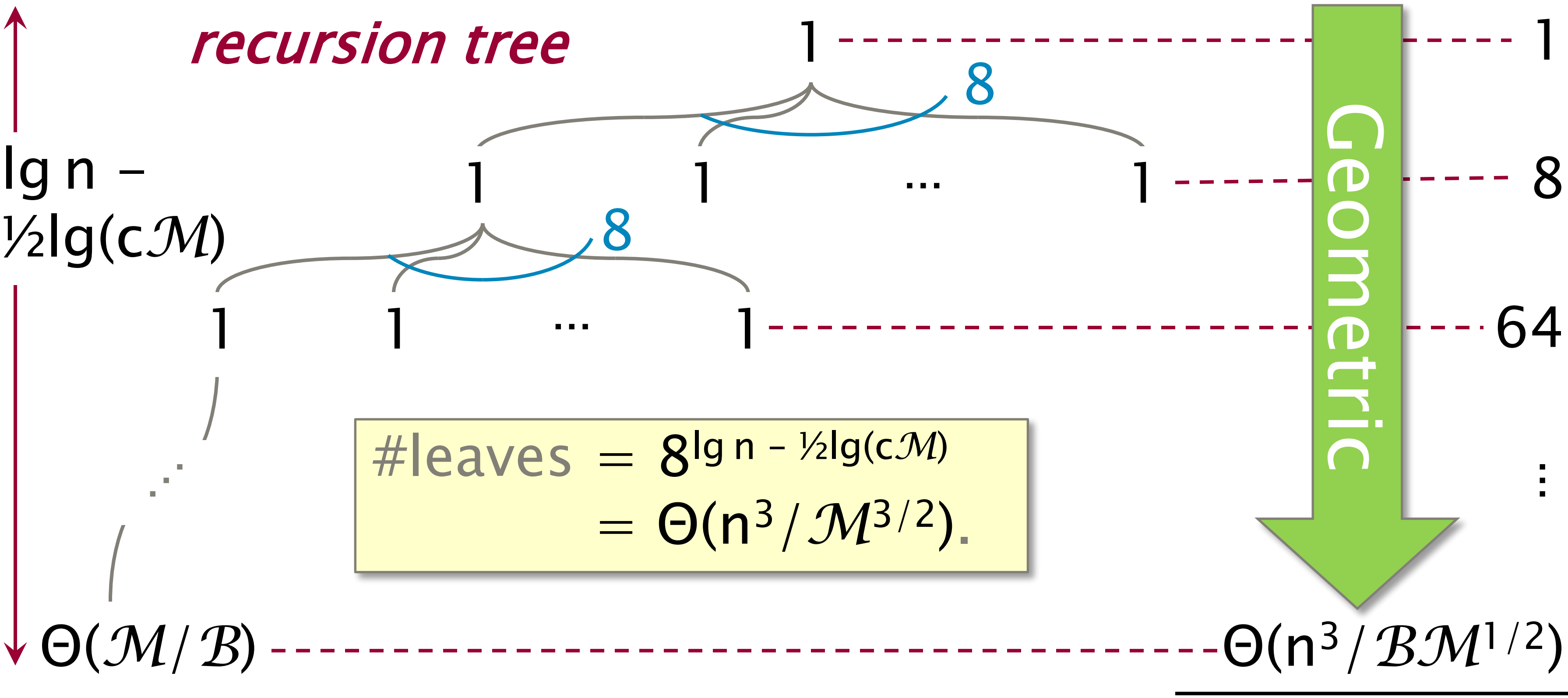
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) & \text{if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

recursion tree



Analysis of Cache Misses

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



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

Same cache misses as with tiling!

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

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`.

6.172 Performance Engineering of Software Systems
Fall 2010