MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

FROM

MODERN ALGORITHMS WORKSHOP

Parallel Algorithms

Prof. Charles E. Leiserson

Dr. Tao B. Schardl

September 19, 2018

Outline

- Introduction
- Cilk Model
- Detecting Nondeterminism
- What Is Parallelism?
- Scheduling Theory Primer
- *Lunch Break*
- Analysis of Parallel Loops
- Case Study: Matrix Multiplication
- Case Study: Jaccard Similarity
- Post-Moore Software

Case Study: Matrix Multiplication

Square-Matrix Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Assume for simplicity that $n = 2^k$.

Matrix Multiplication C Code

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define N 1024
double A[N][N];
double B[N][N];
double C[N][N];

float tdiff(struct timeval *start,
            struct timeval *end) {
  return (end->tv_sec-start->tv_sec) +
    1e-6*(end->tv_usec-start->tv_usec);
}

int main(int argc, const char *argv[]) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      A[i][j] = (double)rand() / (double)RAND_MAX;
      B[i][j] = (double)rand() / (double)RAND_MAX;
      C[i][j] = 0;
    }
  }

  struct timeval start, end;
  gettimeofday(&start, NULL);

  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      for (int k = 0; k < N; ++k) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }

  gettimeofday(&end, NULL);
  printf("%0.6f\n", tdiff(&start, &end));
  return 0;
}
```

The file `mm/mm.c` provides a simple C implementation of this matrix multiplication algorithm.


```c
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    for (int k = 0; k < N; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Exercise: Parallel Matrix Multiply

1. Parallelize the matrix-multiplication code in `mm/mm.c`. Remember to add "`#include <cilk/cilk.h>`" to the top of `mm.c`.
2. Compile and run your code to see its running time:

```
$ cd mm
$ make clean; make
$ ./mm
```

3. Check that your code is correct, and measure its scalability. We've given you a script to make this easier:

```
$ ./mm --verify
$ cilksan ./mm
$ cilkscale ./mm
```

4. Make the code as fast as possible (without calling a matrix-multiplication library).

Parallelizing Matrix Multiply

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

Work: $T_1(n) = \Theta(n^3)$

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

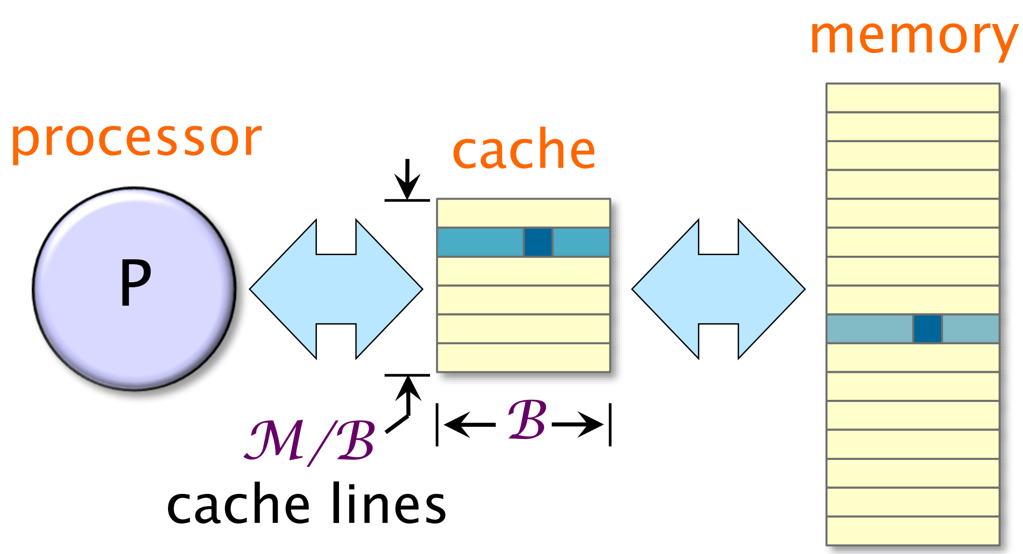
Parallelism: $T_1(n)/T_\infty(n) = \Theta(n^2)$

For 1000×1000 matrices, parallelism $\approx (10^3)^2 = 10^6$.

Hardware Caches, Revisited

IDEA: Restructure the computation to reuse data in the cache as much as possible.

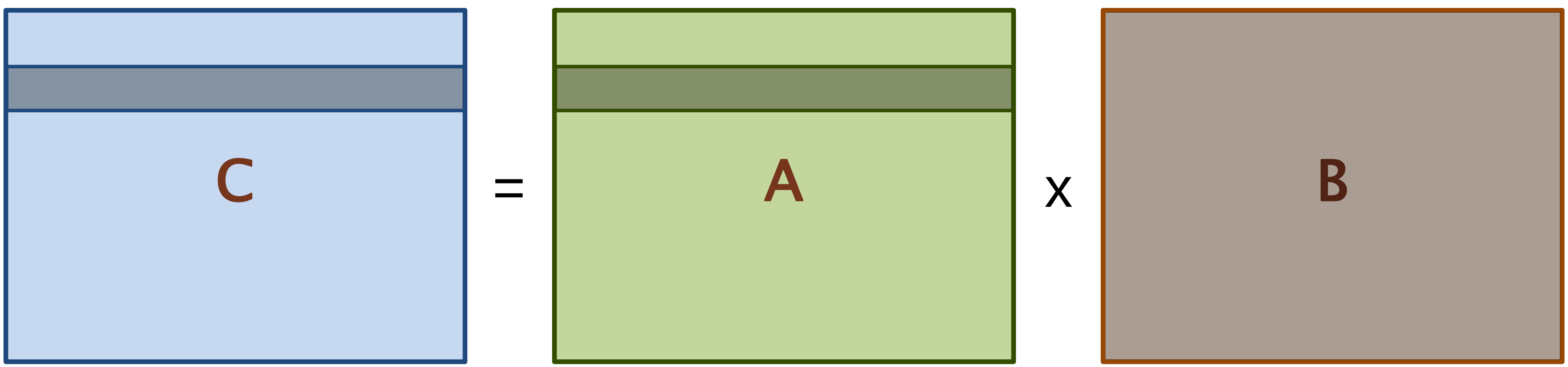
- Cache misses are slow, and cache hits are fast.
- Try to make the most of the cache by reusing the data that's already there.



Data Reuse: Loops

How many memory accesses must the looping code perform to fully compute 1 row of C?

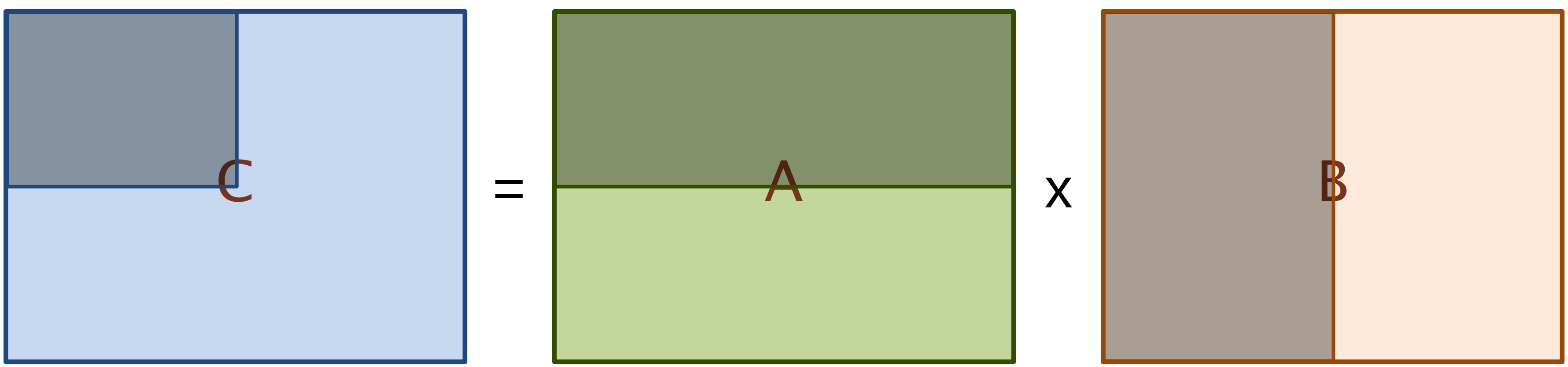
- 1024 * 1 = 1024 writes to C,
- 1024 * 1 = 1024 reads from A, and
- 1024 * 1024 = 1,048,576 reads from B, which is
- 1,050,624 memory accesses total.



Data Reuse: Blocks

How about to compute a 32×32 block of C?

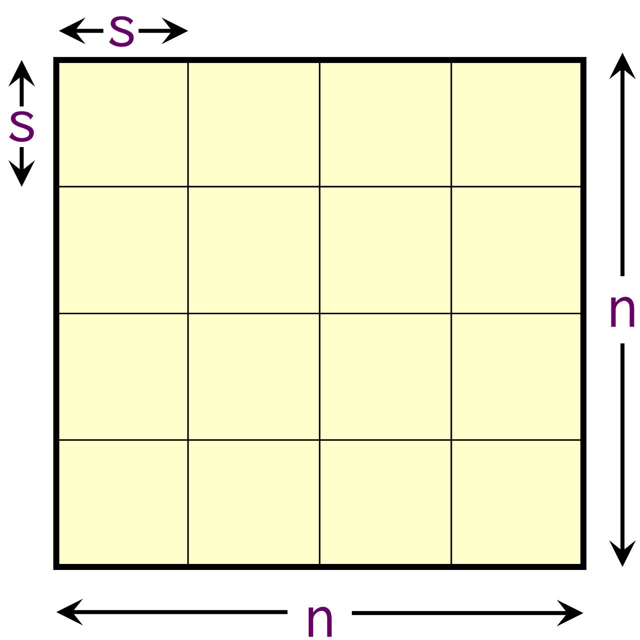
- $32 \cdot 32 = 1024$ writes to C,
- $32 \cdot 1024 = 32{,}768$ reads from A, and
- $1024 \cdot 32 = 32{,}768$ reads from B, or
- $66{,}560$ memory accesses total.



Tiled Matrix Multiplication

```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int il = 0; il < s; ++il)
        for (int kl = 0; kl < s; ++kl)
          for (int jl = 0; jl < s; ++jl)
            C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
```

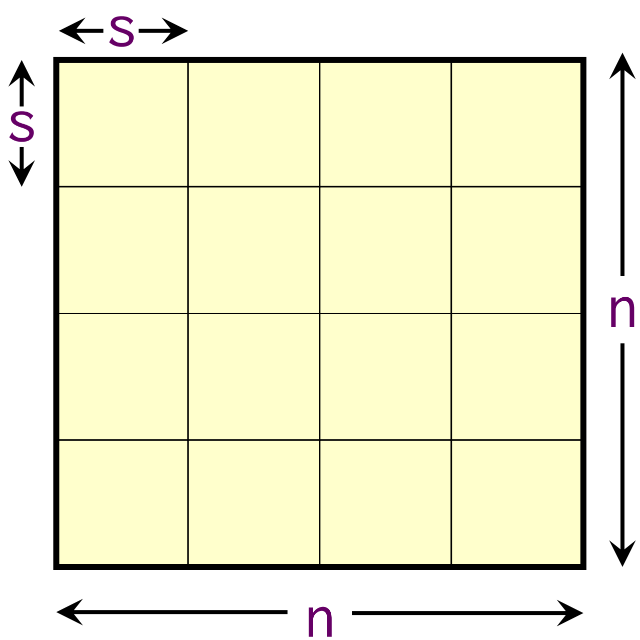
Tuning parameter
How do we find the right value of s? Experiment!



Tile size	Running time (s)
4	0.127
8	0.066
16	0.059
32	0.049
64	0.074
128	0.102

Analysis of Tiled Matrix Multiplication

```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int il = 0; il < s; ++il)
        for (int kl = 0; kl < s; ++kl)
          for (int jl = 0; jl < s; ++jl)
            C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
```



Work: $T_1(n) = \Theta(n^3)$

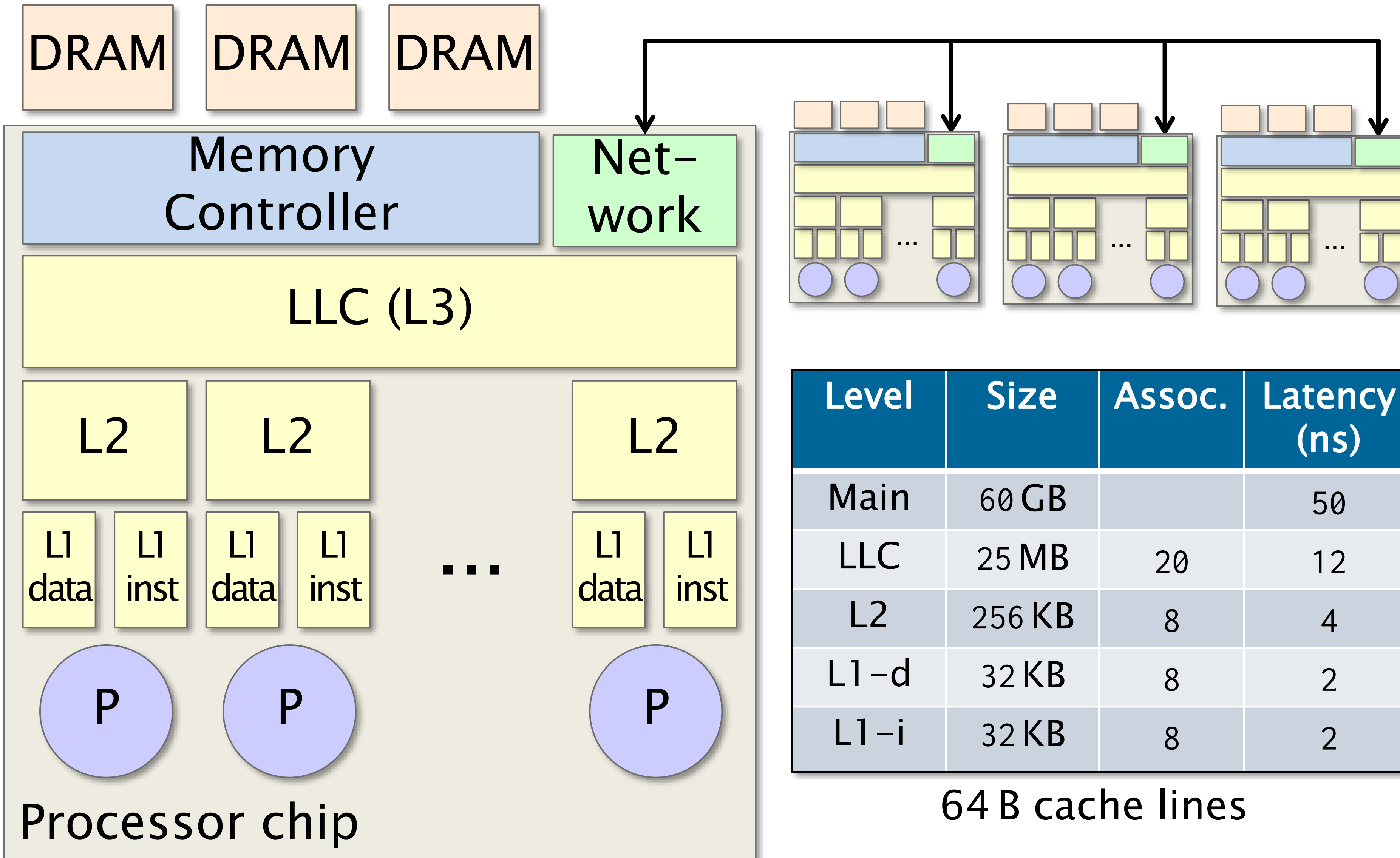
Span: $T_\infty(n) = \Theta(\lg n + (n/s)s^3)$

$= \Theta(ns^2)$

Parallelism:

$T_1(n)/T_\infty(n) = \Theta((n/s)^2)$

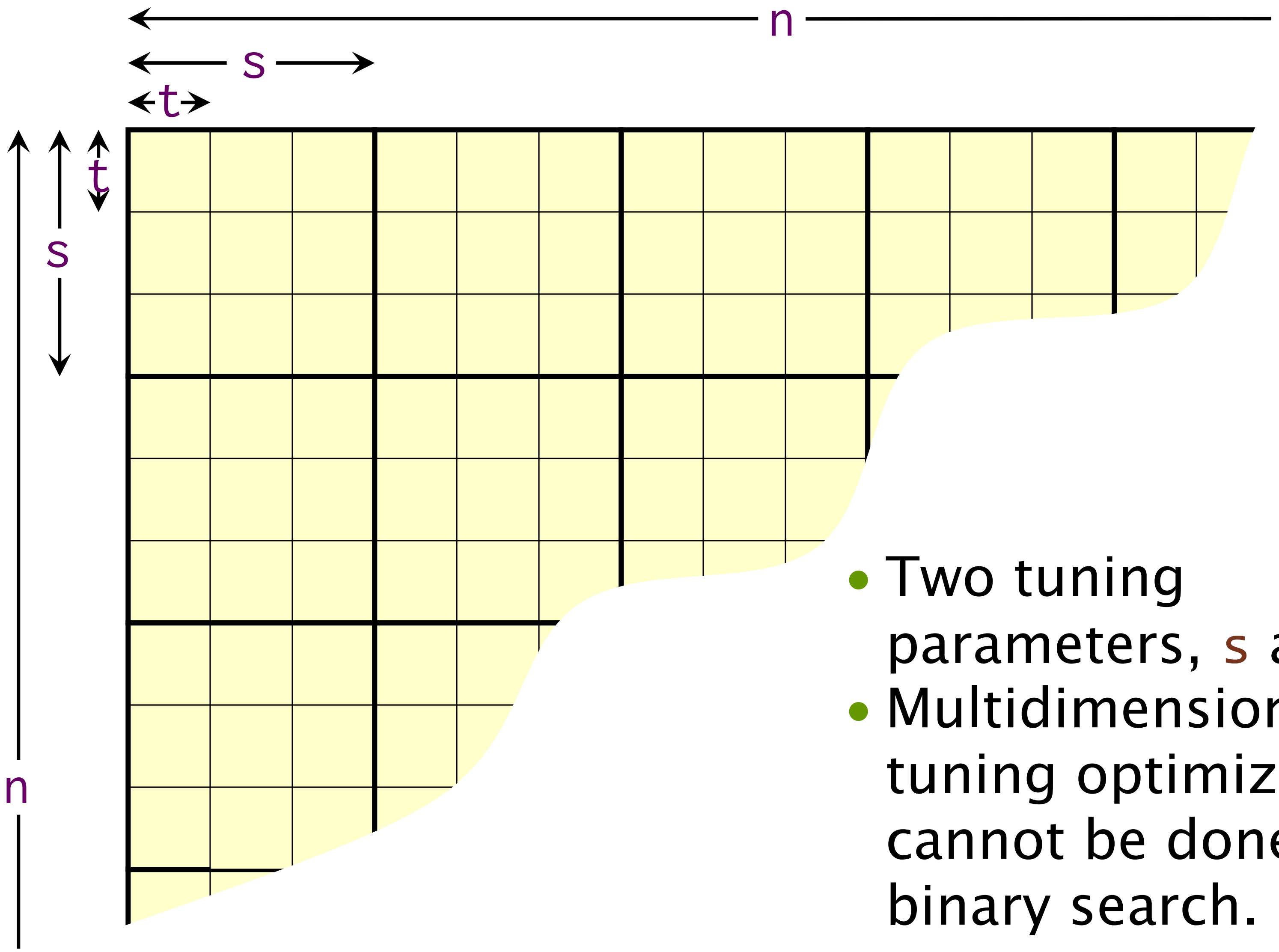
Multicore Cache Hierarchy



Level	Size	Assoc.	Latency (ns)
Main	60 GB		50
LLC	25 MB	20	12
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

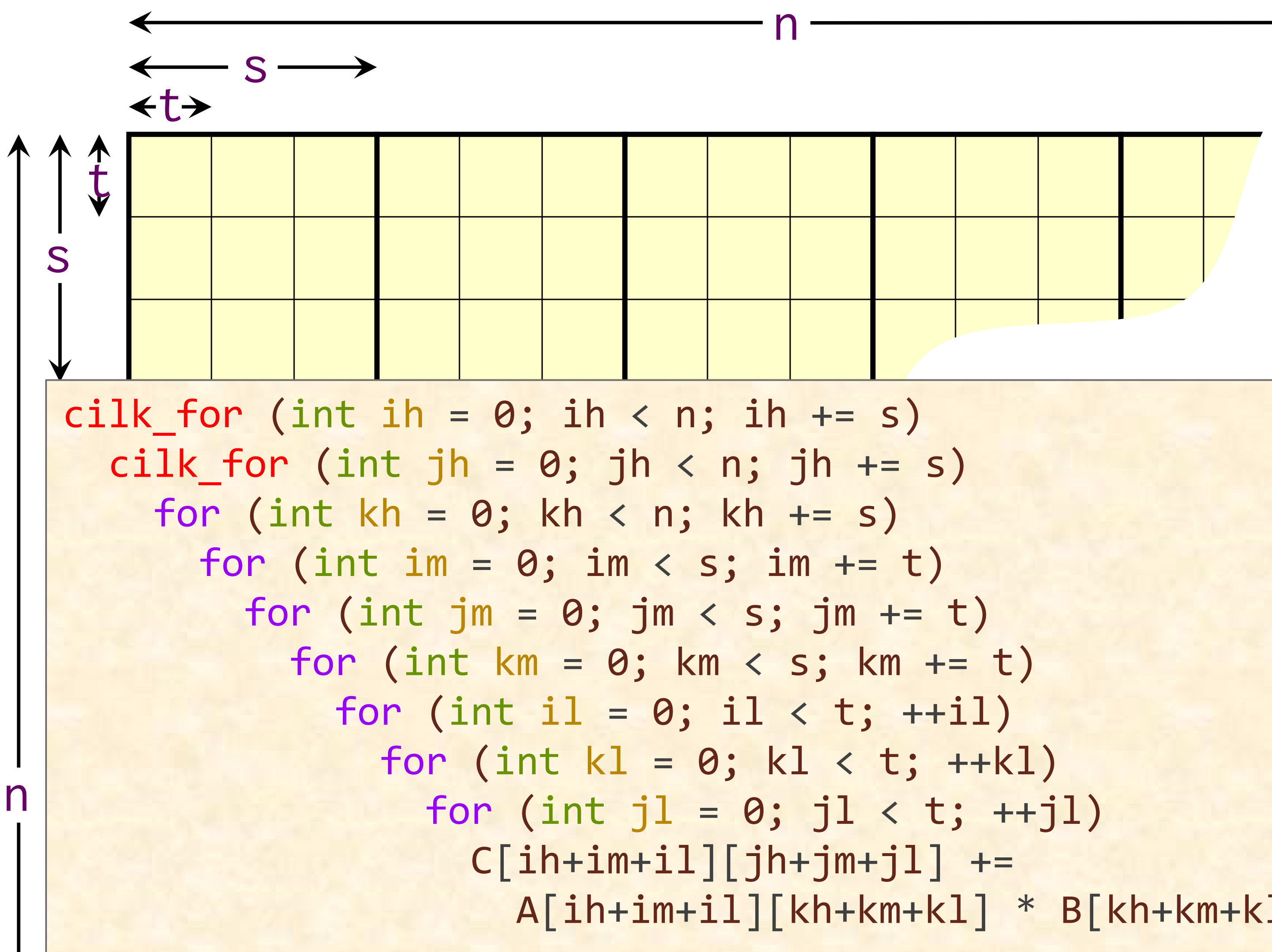
64 B cache lines

Tiling for a Two-Level Cache



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

Tiling for a Two-Level Cache




```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int im = 0; im < s; im += t)
        for (int jm = 0; jm < s; jm += t)
          for (int km = 0; km < s; km += t)
            for (int il = 0; il < t; ++il)
              for (int kl = 0; kl < t; ++kl)
                for (int jl = 0; jl < t; ++jl)
                  C[ih+im+il][jh+jm+jl] +=
                    A[ih+im+il][kh+km+kl] * B[kh+km+kl][jh+jm+jl];
```

Recursive Matrix Multiplication

Divide and conquer — uses cache more efficiently.

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix} + \begin{bmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{bmatrix}$$

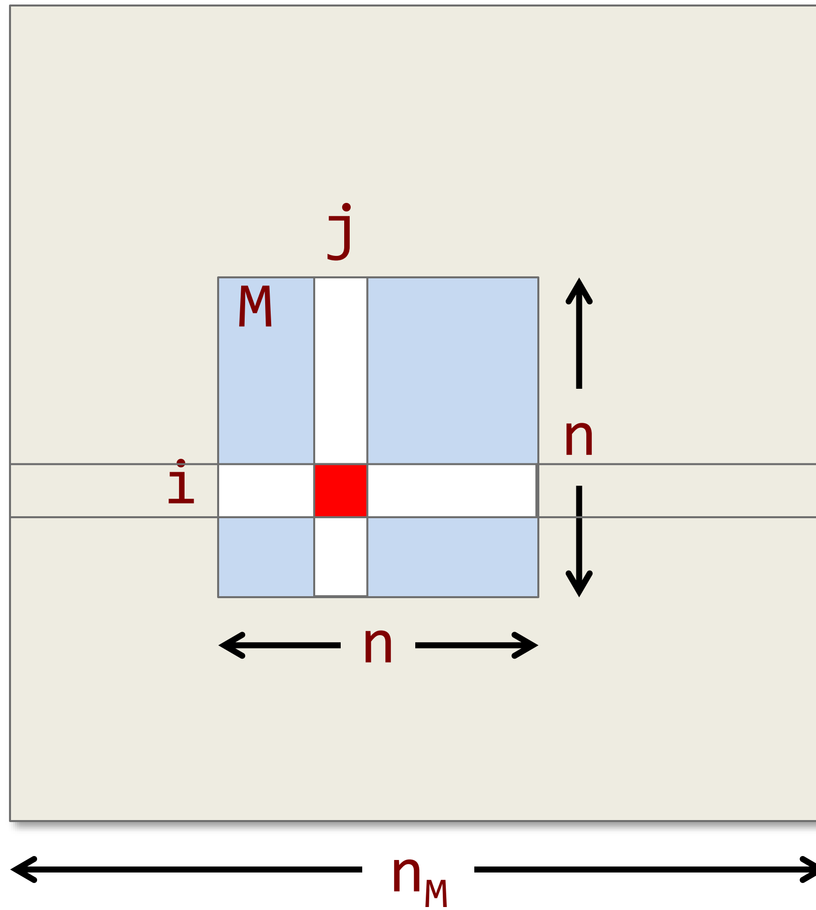
8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.

Representation of Submatrices

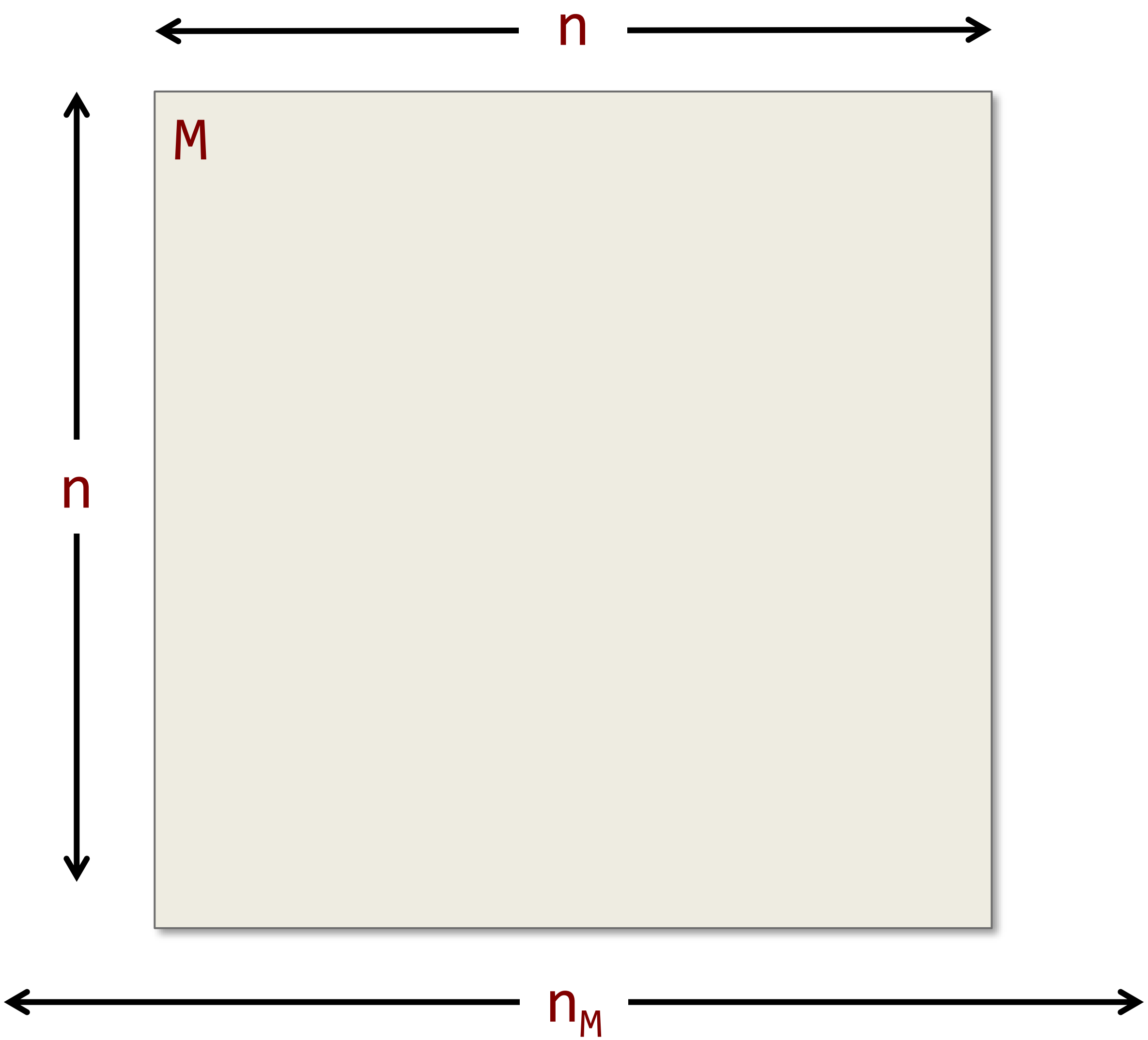
Row-major layout

If M is an $n \times n$ submatrix of an underlying matrix with row size n_M, then the (i,j) element of M is M[n_M*i + j].

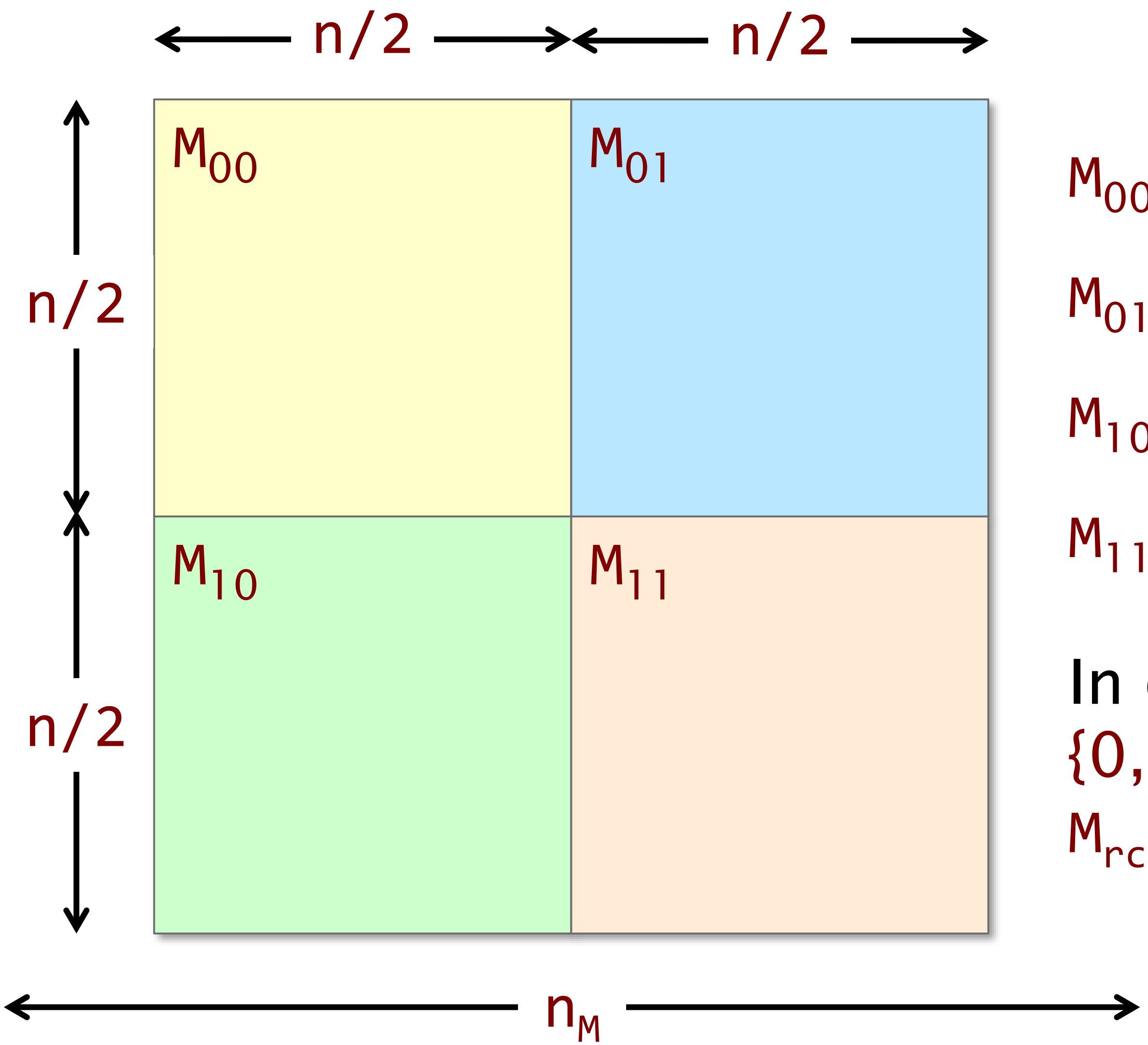
Note: The dimension n does not enter into the calculation.



Divide-and-Conquer Matrices



Divide-and-Conquer Matrices



$M_{00} = M$

$M_{01} = M + (n/2)$

$M_{10} = M + n_M*(n/2)$

$M_{11} = M + (n_M+1)*(n/2)$

In general, for $r,c \in \{0,1\}$, we have

$M_{rc} = M + (r*n_M+c)*(n/2)$

D&C Matrix Multiplication

```c
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
  }
}
```

D&C Matrix Multiplication

```c
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B,
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
  }
}
```

The compiler can assume that the input matrices are not aliased.

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
  }
}
```

The row sizes of the underlying matrices.

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
  }
}
```

The matrices are $n \times n$.

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
  }
}
```

Assert for debugging purposes that n is a power of 2.

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_
            double *restrict A, int n_
            double *restrict B, int n_
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
  }
}
```

Coarsen the leaves of the recursion to lower the overhead for serial execution.

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M
    mm_dac(X(C,0,0
    mm_dac(X(C,0,0
    mm_dac(X(C,0,1
    mm_dac(X(C,0,1
    mm_dac(X(C,1,0
    mm_dac(X(C,1,0
    mm_dac(X(C,1,1
    mm_dac(X(C,1,1
  }
}
```

```
void mm_base(double *restrict C, int n_C,
             double *restrict A, int n_A,
             double *restrict B, int n_B,
             int n)
{ // C = A * B
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int k = 0; k < n; ++k) {
        C[i*n_C + j] += A[i*n_A + k] * B[k*n_B + j];
      }
    }
  }
}
```

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0),       X(B,0,1), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,                   n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,                   n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,                   n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,                   n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,                   n/2);
  }
}
```

A clever macro to compute indices of submatrices.

The C preprocessor's *token-pasting operator*.

D&C Matrix Multiplication

```c
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
  }
}
```

Perform the 8 multiplications of $(n/2)\times(n/2)$ submatrices recursively in parallel.

Exercise: Parallel D&C Matrix Multiply

1. Parallelize the D&C matrix-multiplication code in `mm/mm_dac.c`. Remember to add "`#include <cilk/cilk.h>`" to the top of `mm_dac.c`.
2. Compile and run your code to see its running time:

```
$ make clean; make
$ ./mm_dac
```

3. Check that your code is correct, and measure its scalability. We've given you scripts to make this easier:

```
$ ./mm_dac --verify
$ cilksan ./mm_dac
$ cilkscale ./mm_dac
```

4. Make the code as fast as possible (without calling a matrix-multiplication library).

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Allocate a temporary $n \times n$ array.

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

The temporary array has underlying row size n.

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Perform the 8 multiplications of $(n/2) \times (n/2)$ submatrices recursively in parallel.

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r
    cilk_spawn mm_dac(X(
    cilk_spawn mm_dac(X(
    cilk_spawn mm_dac(X(
    cilk_spawn mm_dac(X(
    cilk_spawn mm_dac(X(
    cilk_spawn mm_dac(X(
    cilk_spawn mm_dac(X(
               mm_dac(X(
    cilk_sync;
    m_add(C, n_C, D, n_D
    free(D);
  }
}
```

```
void mm_base(double *restrict C, int n_C,
             double *restrict A, int n_A,
             double *restrict B, int n_B,
             int n)
{ // C = A * B
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      C[i*n_C + j] = 0;
      for (int k = 0; k < n; ++k) {
        C[i*n_C + j] += A[i*n_A + k] * B[k*n_B + j];
      }
    }
  }
}
```

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Wait for all spawned subcomputations to complete.

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A      n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(     ), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C,    ,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n     (A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0),    , X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1   n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1   ), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D 1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Add the temporary matrix D into the output matrix C.

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) =
  if (n <= THRESHOLD)
    mm_base(C, n_C, A
  } else {
    double *D = mall
    assert(D != NULL)
#define n_D n
#define X(M,r,c) (M +
    cilk_spawn mm_da
    cilk_spawn mm_da
    cilk_spawn mm_da
    cilk_spawn mm_da
    cilk_spawn mm_da
    cilk_spawn mm_da
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

```
void m_add (double *restrict C, int n_C,
            double *restrict D, int n_D,
            int n)
{ // C += D
  cilk_for (int i = 0; i < n; ++i) {
    cilk_for (int j = 0; j < n; ++j) {
      C[i*n_C + j] += D[i*n_D + j];
    }
  }
}
```

Fully Parallel D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Clean up, and then return.

Analysis of Matrix Addition

```
void m_add (double *restrict C, int n_C,
            double *restrict D, int n_D,
            int n)
{ // C += D
  cilk_for (int i = 0; i < n; ++i) {
    cilk_for (int j = 0; j < n; ++j) {
      C[i*n_C + j] += D[i*n_D + j];
    }
  }
}
```

Work: $A_1(n) = \Theta(n^2)$

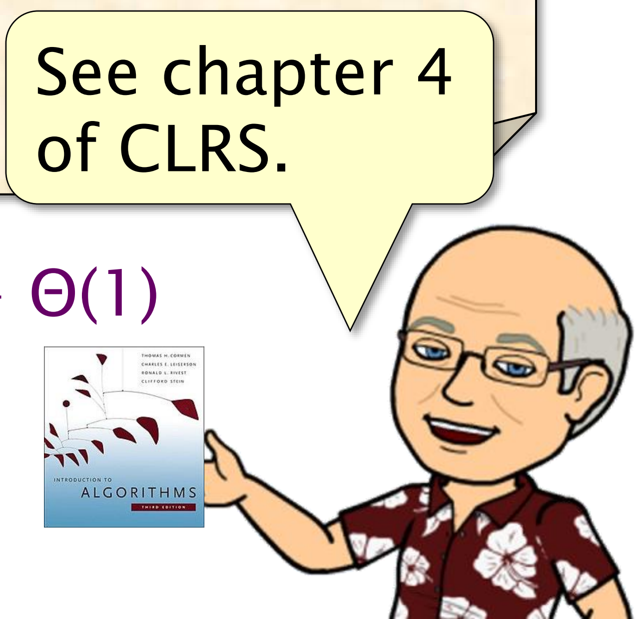
Span: $A_\infty(n) = \Theta(\lg n)$

Work of Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // ...
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Work:

$$
\begin{aligned}
M_1(n) &= 8M_1(n/2) + A_1(n) + \Theta(1) \\
&= 8M_1(n/2) + \Theta(n^2) \\
&= \Theta(n^3)
\end{aligned}
$$



Span of Matrix Multiplication

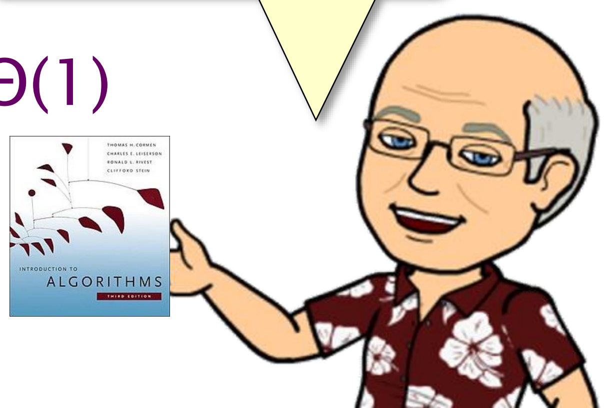
maximum

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // ...
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

See chapter 4 of CLRS.

Span:

$$\begin{aligned} M_\infty(n) &= M_\infty(n/2) + A_\infty(n) + \Theta(1) \\ &= M_\infty(n/2) + \Theta(\lg n) \\ &= \Theta(\lg^2 n) \end{aligned}$$



Parallelism of D&C Matrix Multiply

Work: $M_1(n) = \Theta(n^3)$

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

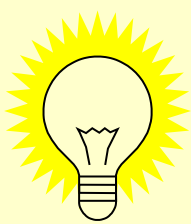
Parallelism: $$\frac{M_1(n)}{M_\infty(n)} = \Theta(n^3/\lg^2 n)$$

For 1000×1000 matrices, parallelism $\approx (10^3)^3/10^2 = 10^7$.

Temporaries

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cil
    ci
    ci
    ci

    ci
    m_
    fr
  }
}
```



IDEA

Since minimizing storage tends to yield higher serial performance, trade off some of the ample parallelism for less storage.

How to Avoid the Temporary?

```c
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

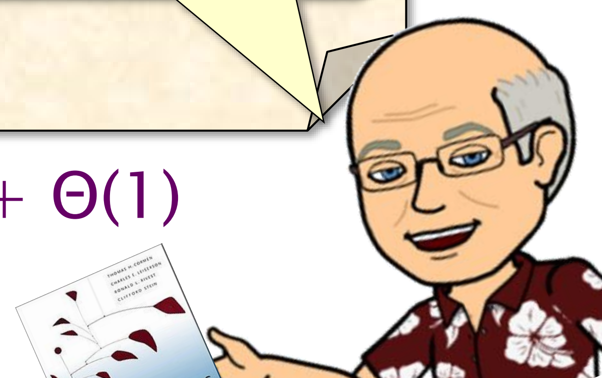
No-Temp Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
  }
}
```

Saves space, but at what expense?

Work of No-Temp Multiply

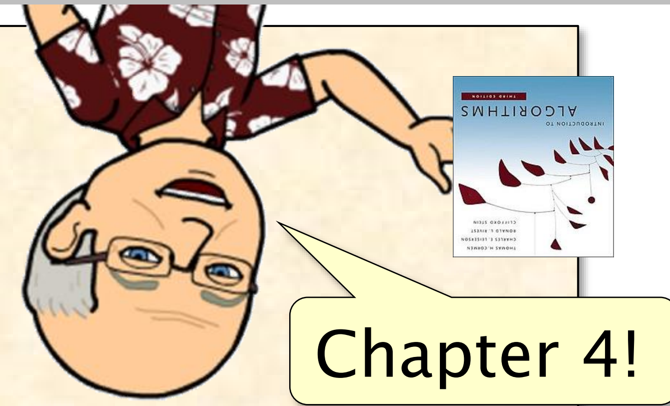
```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n
               mm_dac(X(C,1,1), n_C, X(A,1,1), n
    cilk_sync;
  }
}
```



Work: $M_1(n) = 8M_1(n/2) + \Theta(1)$

$= \Theta(n^3)$

Span of No-Temp Multiply




```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
  }
}
```

max

max

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

Parallelism of No-Temp Multiply

Work: $M_1(n) = \Theta(n^3)$

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

Parallelism: $$\frac{M_1(n)}{M_\infty(n)} = \Theta(n^2)$$

For 1000×1000 matrices,
parallelism $\approx (10^3)^2 = 10^6$.

Faster in practice!