Lecture 8
Analysis of Multithreaded Algorithms
Charles E. Leiserson
DIVIDE-AND-CONQUER RECURRENCES
The Master Method for solving divide-and-conquer recurrences applies to recurrences of the form

\[ T(n) = aT(n/b) + f(n), \]

where \( a \geq 1, b > 1, \) and \( f \) is asymptotically positive.

*The unstated base case is \( T(n) = \Theta(1) \) for sufficiently small \( n. \)
Recursion Tree: \( T(n) = aT(n/b) + f(n) \)
Recursion Tree: \( T(n) = a T(n/b) + f(n) \)
Recursion Tree: \( T(n) = a T(n/b) + f(n) \)
Recursion Tree: \( T(n) = aT(n/b) + f(n) \)
Recursion Tree: $T(n) = aT(n/b) + f(n)$

IDEA: Compare $n^{\log_b a}$ with $f(n)$.

$a^{\log_b n} T(1) = \Theta(n^{\log_b a})$
Master Method — Case I

Specifically, \( f(n) = O(n^{\log_b a - \epsilon}) \) for some constant \( \epsilon > 0 \).

\[
T(n) = \Theta(n^{\log_b a})
\]
Specifically, \( f(n) = \Theta(n^{\log_b a \lg k n}) \) for some constant \( k \geq 0 \).

\[
T(n) = \Theta(n^{\log_b a \lg (k+1)n})
\]
Master Method — Case 3

\[ n^{\log_b a} \ll f(n) \]

**GEOMETRICALLY DEG**

Specifically, \( f(n) = \Omega(n^{\log_b a} + \epsilon) \) for some constant \( \epsilon > 0 \).

\[ h = \log_b n \]

\[ f(n/b^2) \quad f(n/b) \quad f(n) \]

\[ T(1) \]

\[ T(n) = \Theta(f(n)) \]

\[ a \cdot f(n/b) \quad a^2 \cdot f(n/b^2) \]

\[ a^{\log_b n} \cdot T(1) = \Theta(n^{\log_b a}) \]

*and \( f(n) \) satisfies the **regularity condition** that \( a \cdot f(n/b) \leq c \cdot f(n) \) for some constant \( c < 1 \).
Solve

\[ T(n) = a \cdot T(n/b) + f(n) , \]

where \( a \geq 1 \) and \( b > 1 \).

**Case 1:** \( f(n) = O(n^{\log_b a - \varepsilon}) \), constant \( \varepsilon > 0 \)
\[ \Rightarrow T(n) = \Theta(n^{\log_b a}) . \]

**Case 2:** \( f(n) = \Theta(n^{\log_b a \lg^k n}) \), constant \( k \geq 0 \)
\[ \Rightarrow T(n) = \Theta(n^{\log_b a \ lg^{k+1} n}) . \]

**Case 3:** \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \), constant \( \varepsilon > 0 \)
(and regularity condition)
\[ \Rightarrow T(n) = \Theta(f(n)) . \]
Master Method Quiz

- \( T(n) = 4T(n/2) + n \)
  \( n^{\log_b a} = n^2 \gg n \Rightarrow \text{Case 1: } T(n) = \Theta(n^2). \)

- \( T(n) = 4T(n/2) + n^2 \)
  \( n^{\log_b a} = n^2 = n^2 \lg^0 n \Rightarrow \text{Case 2: } T(n) = \Theta(n^2 \lg n). \)

- \( T(n) = 4T(n/2) + n^3 \)
  \( n^{\log_b a} = n^2 \ll n^3 \Rightarrow \text{Case 3: } T(n) = \Theta(n^3). \)

- \( T(n) = 4T(n/2) + n^2/\lg n \)
  Master method does not apply!
  Answer is \( T(n) = \Theta(n^2 \lg \lg n). \) (Prove by substitution.)

More general (but more complicated) solution: Akra–Bazzi method.
Cilk Loops
Loop Parallelism in Cilk

Example: In-place matrix transpose

\[ \begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix} \rightarrow
\begin{pmatrix}
  a_{11} & a_{21} & \cdots & a_{n1} \\
  a_{12} & a_{22} & \cdots & a_{n2} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{1n} & a_{2n} & \cdots & a_{nn}
\end{pmatrix} \]

A \rightarrow A^T

The iterations of a \texttt{cilk\_for} loop execute in parallel.

```c
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i)
  { for (int j=0; j<i; ++j)
      { double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
      }
  }
```

© 2008–2018 by the MIT 6.172 Lecturers
Implementation of Parallel Loops

```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

void recur(int lo, int hi)  // half open
{
    if (hi > lo + 1) {
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid, hi);
        cilk_sync;
        return;
    }
    int i = lo;
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
::
recur(1, n);
```

Divide–and–conquer implementation

The Tapir/LLVM compiler implements `cilk_for` loops this way at optimization level `-01` or higher.
Implementation of Parallel Loops

```c
// indices run from 0, not 1
void recur(int lo, int hi) { // half open
    if (hi > lo + 1) {
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
            recur(mid, hi);
        cilk_sync;
        return;
    }
    int i = lo;
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
    ...
    recur(1, n);
}
```
Execution of Parallel Loops

```c
void recur(int lo, int hi) //half open
{
    if (hi > lo + 1) {
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid, hi);
        cilk_sync;
        return;
    }
    int i = lo;
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

**Divide-and-conquer implementation**

**cilk_for loop control**

**Computation dag**
Analysis of Parallel Loops

// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}

Span of loop control
= Θ(lgn) .
Max span of body
= Θ(n) .

Work: \( T_1(n) = \Theta(n^2) \)
Span: \( T_\infty(n) = \Theta(n + \log n) = \Theta(n) \)
Parallelism: \( T_1(n)/T_\infty(n) = \Theta(n) \)

© 2008–2018 by the MIT 6.172 Lecturers
Analysis of Nested Parallel Loops

```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

Span of outer loop control = $\Theta(\lg n)$. 

Max span of inner loop control = $\Theta(\lg n)$. 

Span of body = $\Theta(1)$. 

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

Span: $T_{\infty}(n) = \Theta(\lg n)$

Parallelism: $T_1(n)/T_{\infty}(n) = \Theta(n^2/\lg n)$
A Closer Look at Parallel Loops

```cilk_for (int i=0; i<n; ++i) {
  A[i] += B[i];
}
```

**Vector addition**

- **Work**: $T_1 = \Theta(n)$
- **Span**: $T_\infty = \Theta(\lg n)$
- **Parallelism**: $T_1 / T_\infty = \Theta(n / \lg n)$

Includes substantial overhead!
Coarsening Parallel Loops

```c
#pragma cilk grainsize G
cilk_for (int i=0; i<n; ++i) {
    A[i] += B[i];
}
```

If a grainsize pragma is not specified, the Cilk runtime system makes its best guess to minimize overhead.

```c
void recur(int lo, int hi) { //half open
    if (hi > lo + G) {
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid, hi);
        cilk_sync;
        return;
    }
    for (int i=lo; i<hi; ++i) {
        A[i] += B[i];
    }
}
```

Implementation with coarsening

```c
void recur(0, n);
```
Let $I$ be the time for one iteration of the loop body. Let $S$ be the time to perform a spawn and return.
Loop Grain Size

Vector addition

```c
#pragma cilk grainsize G
cilk_for (int i=0; i<n; ++i) {
    A[i] += B[i];
}
```

Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

Span: $T_\infty = G \cdot I + \lg(n/G) \cdot S$

Want $G \gg S/I$ and $G$ small.
Another Implementation

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

for (int j=0; j<n; j+=G) {
    cilk_spawn vadd(A+j, B+j, MIN(G,n-j));
}

} cilk_sync;
```

---

Assume that \( G = 1 \).

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

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

**Parallelism:** \( T_1 / T_\infty = \Theta(1) \)
Another Implementation

```c
void vadd (double *A, double *B, int n){
    for (int i=0; i<n; i++) A[i] += B[i];
    ...
    for (int j=0; j<n; j+=G) {
        cilk_spawn vadd(A+j, B+j, MIN(G,n-j));
    }
    cilk_sync;
}
```

Choose $G = \sqrt{n}$ to minimize.

Analyze in terms of $G$:

Work: $T_1 = \Theta(n)$

Span: $T_{\infty} = \Theta(G + n/G) = \Theta(\sqrt{n})$

Parallelism: $T_1/T_{\infty} = \Theta(\sqrt{n})$
**Quiz on Parallel Loops**

**Question:** Let $P \ll n$ be the number of workers on the system. How does the parallelism of Code A compare to the parallelism of Code B? (Differences highlighted.)

**Code A**

```c
#pragma cilk grainsize 1
cilk_for (int i=0; i<n; i+=32) {
    for (int j=i; j<MIN(i+32, n); ++j)
        A[j] += B[j];
}
```

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

**Span:** $T_\infty = \Theta(lg(n/32) + 32)$

$\quad = \Theta(lg n)$

**Parallelism:**

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

**Code B**

```c
#pragma cilk grainsize 1
cilk_for (int i=0; i<n; i+=n/P) {
    for (int j=i; j<MIN(i+n/P, n); ++j)
        A[j] += B[j];
}
```

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

**Span:** $T_\infty = \Theta(lg P + n/P)$

$\quad = \Theta(n/P)$

**Parallelism:**

$$T_1/T_\infty = \Theta(P)$$
Quiz on Parallel Loops

Question: Let \( P \ll n \) be the number of workers on the system. How does the parallelism of Code A compare to the parallelism of Code B? (Differences highlighted.)

**Code A**

```cilk
#pragma cilk grainsize 1

cilk_for (int i=0; i<n; i+=32) {
    for (int j=i; j<\min(i+32, n); ++j)
        A[j] += B[j];
}
```

**Code B**

```cilk
#pragma cilk grainsize 1

cilk_for (int i=0; i<n; i+={n/P}) {
    for (int j=i; j<\min(i+n/P, n); ++j)
        A[j] += B[j];
}
```

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

**Span**: \( T_\infty = \Theta(\lg(n/32) + 32) = \Theta(\lg n) \)

**Parallelism**: \( T_1/T_\infty \)

**Want** \( T_1/T_\infty \gg P \).

**Code A**

```cilk
#pragma cilk grainsize 1

cilk_for (int i=0; i<n; i+=32) {
    for (int j=i; j<\min(i+32, n); ++j)
        A[j] += B[j];
}
```

**Code B**

```cilk
#pragma cilk grainsize 1

cilk_for (int i=0; i<n; i+={n/P}) {
    for (int j=i; j<\min(i+n/P, n); ++j)
        A[j] += B[j];
}
```

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

**Span**: \( T_\infty = \Theta(\lg P + n/P) = \Theta(n/P) \)

**Parallelism**: \( T_1/T_\infty = \Theta(P) \)
Three Performance Tips

1. **Minimize the span** to maximize parallelism. Try to generate 10 times more parallelism than processors for near-perfect linear speedup.

2. If you have plenty of parallelism, try to trade some of it off to reduce **work overhead**.

3. Use **divide-and-conquer recursion** or **parallel loops** rather than spawning one small thing after another.

**Do this:**
```cilk
for (int i=0; i<n; ++i) {
    foo(i);
}
```

**Not this:**
```cilk
for (int i=0; i<n; ++i) {
    cilk_spawn foo(i);
    cilk_sync;
}
```
And Three More

4. Ensure that work/#spawns is sufficiently large.
   - Coarsen by using function calls and inlining near the leaves of recursion, rather than spawning.

5. Parallelize outer loops, as opposed to inner loops, if you’re forced to make a choice.

6. Watch out for scheduling overheads.

   Do this:
   ```cilk
   cilk_for (int i=0; i<2; ++i) {
     for (int j=0; j<n; ++j)
       f(i,j);
   }
   ```

   Not this:
   ```cilk
   for (int j=0; j<n; ++j) {
     cilk_for (int i=0; i<2; ++i)
       f(i,j);
   }
   ```
MATRIX MULTIPLICATION
Square-Matrix Multiplication

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

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

Assume for simplicity that \( n = 2^k \).
Parallelizing Matrix Multiply

```cilk
  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 $1000 \times 1000$ matrices, parallelism $\approx (10^3)^2 = 10^6$. 
Divide and conquer — uses cache more efficiently, as we’ll see later in the term.

\[
\begin{pmatrix}
C_{00} & C_{01} \\
C_{10} & C_{11}
\end{pmatrix} =
\begin{pmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{pmatrix} \cdot
\begin{pmatrix}
B_{00} & B_{01} \\
B_{10} & B_{11}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
A_{00}B_{00} & A_{00}B_{01} \\
A_{10}B_{00} & A_{10}B_{01}
\end{pmatrix} +
\begin{pmatrix}
A_{01}B_{10} & A_{01}B_{11} \\
A_{11}B_{10} & A_{11}B_{11}
\end{pmatrix}
\]

8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.
Row-major layout

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

Note: The dimension $n$ does not enter into the calculation, although it does matter for bounds checking of $i$ and $j$. 
Divide-and-Conquer Matrices

A

\[ n \]

\[ n_M \]

\[ \ldots \]

\[ n_M \]

\[ \ldots \]
Divide-and-Conquer Matrices

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

$$A_{rc} = A + (r n_M + c)(n/2)$$
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);
}

© 2008–2018 by the MIT 6.172 Lecturers
The compiler can assume that the input matrices are not aliased.
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 {
        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);
        cilk_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), 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 row sizes of the underlying matrices.
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);
        cilk_spawn 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 function adds the matrix product $A \times B$ to matrix $C$. 

```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 \times 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,0,0), 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,0,0), 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,1,0), 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 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);
        cilk_spawn 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 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);
    }
}
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 {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);
        #define n_D n
        #define X(M,r,c) (M + (r)*(c))
        cilk_spawn mm_dac(X(n, n_D, 0));
        cilk_spawn mm_dac(X(n, n_D, 1));
        cilk_spawn mm_dac(X(n, n_D, 2));
        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) {
            for (int k = 0; k < n; ++k) {
                C[i*n_C + j] += A[i*n_A + k] * B[k*n_B + j];
            }
        }
    }
}
```

Coarsen the leaves of the recursion to lower the overhead.
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 {
        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,1,0), 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);
    }
    m_add(C, n_C, D, n_D, n);
    free(D);
}
```

Allocate a temporary $n \times n$ array D.
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 {
        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);
        cilk_spawn 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 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,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,1), n_C, X(A,0,0), 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), X(B,1,0), n_B, n/2);
        cilk_spawn mm_dac(X(D,1,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        cilk_spawn mm_dac(X(D,1,1), n_D, X(A,0,1), n_A, X(B,1,0), 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.
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 {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);
        #define n_D n
        #define X(M, r, c) (M + (r*(n_ ## M) + c)*)
        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,0,0), 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,1,0), n_A, X(B,1,0), n_B, n/2);
        cilk_spawn 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.
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 {
        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_D))
        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,0,1), 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,1,1), 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,1,0), n_A, X(B,1,0), n_B, n/2);
        cilk_spawn 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.
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_ ## M))
    cilk_spawn mm_dac(X(C,0,0), X(A,0,0), X(B,0,0), n_B, n_A, n_B, n_A, n);
    cilk_spawn mm_dac(X(C,0,1), X(A,0,1), X(B,0,1), n_B, n_A, n_B, n_A, n);
    cilk_spawn mm_dac(X(C,1,0), X(A,1,0), X(B,1,0), n_B, n_A, n_B, n_A, n);
    cilk_spawn mm_dac(X(C,1,1), X(A,1,1), X(B,1,1), n_B, n_A, n_B, n_A, n);
    cilk_spawn mm_dac(X(D,0,0), X(A,0,0), X(B,0,0), n_B, n_A, n_B, n_A, n);
    cilk_spawn mm_dac(X(D,0,1), X(A,0,1), X(B,0,1), n_B, n_A, n_B, n_A, n);
    cilk_spawn mm_dac(X(D,1,0), X(A,1,0), X(B,1,0), n_B, n_A, n_B, n_A, n);
    cilk_spawn mm_dac(X(D,1,1), X(A,1,1), X(B,1,1), n_B, n_A, n_B, n_A, n);
  }
  cilk_sync;
  m_add(C, n_C, D, n_D, n);
  free(D);
}

Add the temporary matrix D into the output matrix C.
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 {
        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_ ## M))
        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);
        cilk_spawn 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);
```
Analysis of Matrix Addition

```c
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

```c
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_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
}
```

**Case 1**

\[ n^{\log_b a} = n^{\log_2 8} = n^3 \]

\[ f(n) = \Theta(n^2) \]

**Work:**

\[ M_1(n) = 8M_1(n/2) + A_1(n) + \Theta(1) \]

\[ = 8M_1(n/2) + \Theta(n^2) \]

\[ = \Theta(n^3) \]
Span of Matrix Multiplication

\[
M_\infty(n) = M_\infty(n/2) + A_\infty(n) + \Theta(1)
\]

\[
= M_\infty(n/2) + \Theta(\lg n)
\]

\[
= \Theta(\lg^2 n)
\]

**CASE 2**
\[
n^{\log_b a} = n^{\log_2 1} = 1
\]
\[
f(n) = \Theta(n^{\log_b a \lg 1 n})
\]
Parallelism of 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 \(1000 \times 1000\) matrices, parallelism \(\approx (10^3)^3/10^2 = 10^7\).
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
    }
}

Since minimizing storage tends to yield higher 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);
    }
    m_add(C, n_C, D, n_D, n);
    free(D);
}```
 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) / 4)) * (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_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);
        cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        cilk_sync;
    }
}
No-Temp 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))
        cilk_spawn mm_dac(X(0,0,0), n_C, X(0,0,0), n_A, X(0,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(0,0,1), n_C, X(0,0,0), n_A, X(0,0,1), n_B, n/2);
        cilk_spawn mm_dac(X(0,1,0), n_C, X(0,1,0), n_A, X(0,0,0), n_B, n/2);
        mm_dac(X(1,1,0), n_C, X(1,1,0), n_A, X(0,1,0), n_B, n/2);
        cilk_sync;
        mm_dac(X(1,0,0), n_C, X(0,1,0), n_A, X(1,0,0), n_B, n/2);
        mm_dac(X(0,1,1), n_C, X(0,1,1), n_A, X(0,1,0), n_B, n/2);
        mm_dac(X(1,1,1), n_C, X(1,1,1), n_A, X(1,1,0), n_B, n/2);
        cilk_sync;
    }
}
```

Saves space, but at what expense?
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,0,0), n_B, n/2);

cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,0,1), n_B, n/2);

cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,0,0), n_B, n/2);
        mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,0,1), n_B, n/2);

cilk_sync;
}

CASE 1
\[ n^{\log_b \alpha} = n^{\log_2 8} = n^3 \]
\[ f(n) = \Theta(1) \]

Work:
\[ M_1(n) = 8M_1(n/2) + \Theta(1) \]
\[ = \Theta(n^3) \]
Span of No-Temp Multiply

```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))
        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),
                mm_dac(X(C,1,0), n_C, X(A,0,1),
                        mm_dac(X(C,0,1), n_C, X(A,0,1),
                                mm_dac(X(C,1,0), n_C, X(A,1,1),
                                        mm_dac(X(C,1,1), n_C, X(A,1,1),
                                                cilk_sync;
                                        )
                                )
                        )
                    )
                )
            )
        )
    }
}
```

**Case 1**

\[ n^{\log_b a} = n^{\log_2 2} = n \]

\[ f(n) = \Theta(1) \]

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

Faster in practice!
MERGE SORT
void merge(int *C, int *A, int na, int *B, int nb) {
    while (na>0 && nb>0) {
        if (*A <= *B) {
            *C++ = *A++; na--;
        } else {
            *C++ = *B++; nb--;
        }
    }
    while (na>0) {
        *C++ = *A++; na--;
    }
    while (nb>0) {
        *C++ = *B++; nb--;
    }
}
void merge_sort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int C[n];
        cilk_spawn merge_sort(C, A, n/2);
        merge_sort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        merge(B, C, n/2, C+n/2, n-n/2);
    }
}
**Merge Sort**

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

19  3  12  46  33  4  21  14

© 2008–2018 by the MIT 6.172 Lecturers
# Merge Sort

```c
void merge_sort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int C[n];
        cilk_spawn merge_sort(C, A, n/2);
        merge_sort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        merge(B, C, n/2, C+n/2, n-n/2);
    }
}
```
Merge Sort

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

merge
merge_sort

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

merge
void merge_sort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int C[n];
        cilk_spawn merge_sort(C, A, n/2);
        merge_sort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        merge(B, C, n/2, C+n/2, n-n/2);
    }
}
Work of Merge Sort

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

**Work:** 
\[
T_1(n) = 2T_1(n/2) + \Theta(n) = \Theta(n \log n)
\]

**Case 2**
\[
n^{\log_b a} = n^{\log_2 2} = n
f(n) = \Theta(n^{\log_b a} \log^0 n)
\]
Span of Merge Sort

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

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

\[ = \Theta(n) \]

**CASE 3**  
\( n^{\log_b a} = n^{\log_2 1} = 1 \)  
f(n) = \( \Theta(n) \)
Parallelism of Merge Sort

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

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

**Parallelism:** \( \frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n) \)

We need to parallelize the merge!
KEY IDEA: If the total number of elements to be merged in the two arrays is \( n = na + nb \), the total number of elements in the larger of the two recursive merges is at most \((3/4)n\).
Coarsen base cases for efficiency.
Span of Parallel Merge

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

**Case 2**

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

\[ f(n) = \Theta(n^{\log_b a \lg n}) \]

\[ \text{Span: } T_\infty(n) \leq T_\infty(3n/4) + \Theta(\lg n) = \Theta(\lg^2 n) \]
Work of Parallel Merge

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

**Work:** \( T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(lg n) \),
where \( 1/4 \leq \alpha \leq 3/4 \).

**Claim:** \( T_1(n) = \Theta(n) \).
Analysis of Work Recurrence

\[ T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n), \]
where \(1/4 \leq \alpha \leq 3/4\).

Substitution method: Inductive hypothesis is \(T_1(k) \leq c_1k - c_2\lg k\), where \(c_1, c_2 > 0\). Prove that the relation holds, and solve for \(c_1\) and \(c_2\).

\[ T_1(n) \leq c_1(\alpha n) - c_2\lg(\alpha n) + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n) \]
\[ \leq c_1n - c_2\lg(\alpha n) - c_2\lg((1-\alpha)n) + \Theta(\lg n) \]
\[ \leq c_1n - c_2\left(\lg(\alpha(1-\alpha)) + 2\lg n\right) + \Theta(\lg n) \]
\[ \leq c_1n - c_2\lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \]
\[ \leq c_1n - c_2\lg n, \]
by choosing \(c_2\) large enough. Choose \(c_1\) large enough to handle the base case.
Parallelism of Parallel Merge

**Work:** \[ T_1(n) = \Theta(n) \]

**Span:** \[ T_\infty(n) = \Theta(\lg^2 n) \]

**Parallelism:** \[ \frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n) \]
Parallel Merge Sort

void p_merge_sort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int C[n];
        cilk_spawn p_merge_sort(C, A, n/2);
        p_merge_sort(C+n/2, A+n/2, n/2);
        cilk_sync;
        p_merge(B, C, n/2, C+n/2, n);
    }
}

CASE 2

\[ n^{\log_b a} = n^{\log_2 2} = n \]

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

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

\[ = \Theta(n \lg n) \]
Parallel Merge Sort

```c
void p_merge_sort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int C[n];
        cilk_spawn p_merge_sort(C, A, n/2);
        p_merge_sort(C+n/2, A+n/2, n/2);
        cilk_sync;
        p_merge(B, C, n/2, C+n/2, n);
    }
}
```

**CASE 2**

\[
n_{\log_b a} = n_{\log_2 1} = 1
\]

\[
f(n) = \Theta(n^{\log_b a} \log^2 n)
\]

**Span:**

\[
T_\infty(n) = T_\infty(n/2) + \Theta(\log^2 n)
\]

\[
= \Theta(\log^3 n)
\]
Parallelism of P_MergeSort

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

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

**Parallelism:** \( \frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n) \)
TABLEAU CONSTRUCTION
Constructing a Tableau

Problem: Fill in an $n \times n$ tableau $A$, where
$A[i][j] = f(A[i][j-1], A[i-1][j], A[i-1][j-1])$.

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

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

Work: $\Theta(n^2)$.

© 2008–2018 by the MIT 6.172 Lecturers
Recursive Construction

Parallel code

I;
cilk_spawn II;
III;
cilk_sync;
IV;
**Recursive Construction**

**Parallel code**

```plaintext
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

*CASE 1*

\[ n^{\log_b a} = n^{\log_2 4} = n^2 \]

\[ f(n) = \Theta(1) \]

*Work:*

\[ T_1(n) = 4T_1(n/2) + \Theta(1) \]

\[ = \Theta(n^2) \]
Recursive Construction

Parallel code

I;
cilk_spawn II;
III;
cilk_sync;
IV;

Case 1
\[ n^{\log_b a} = n^{\log_2 3} = n^{\lg 3} \]
\[ f(n) = \Theta(1) \]

\[ \text{Span: } T_\infty(n) = 3T_\infty(n/2) + \Theta(1) \]
\[ = \Theta(n^{\lg 3}) \]
Analysis of Tableau Constr.

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

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

**Parallelism:** \[
\frac{T_1(n)}{T_\infty(n)} = \Theta(n^{2-\lg 3})
\]
\[
= \Omega(n^{0.41})
\]
## A More–Parallel Construction

### Work:

\[ T_1(n) = 9T_1(n/3) + \Theta(1) \]

\[ = \Theta(n^2) \]

---

**CASE 1**

\[ n^{\log_b a} = n^{\log_3 9} = n^2 \]

\[ f(n) = \Theta(1) \]
# A More–Parallel Construction

\[ n \log_{b^a} n = n \log_{3^5} n \]

**Span:** \( T_\infty(n) = 5T_\infty(n/3) + \Theta(1) = \Theta(n \log_3 5) \)

**Case 1**

\[ n^{\log_{b^a} n} = n^{\log_{3^5} n} \]

\[ f(n) = \Theta(1) \]
Analysis of Revised Method

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

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

**Parallelism:**
\[
\frac{T_1(n)}{T_\infty(n)} = \Theta(n^{2-\log_3 5}) = \Omega(n^{0.53})
\]

Nine-way divide-and-conquer has about \( \Theta(n^{0.12}) \) more parallelism than four-way divide-and-conquer, but it exhibits less cache locality.
What is the largest parallelism that can be obtained for the tableau-construction problem using pure Cilk?

- You may only use basic fork-join control constructs (cilk_spawn, cilk_sync, cilk_for) for synchronization.
- No using locks, atomic instructions, synchronizing through memory, etc.