LECTURE 7
Analysis of Multithreaded Algorithms
Charles E. Leiserson
September 25, 2014
DIVIDE–AND–CONQUER 
RECURRENCES
The Master Method for solving divide–and–conquer recurrences applies to recurrences of the form\(^*\)

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

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: \[ T(n) = a T(n/b) + f(n) \]
Recursion Tree: $T(n) = a \cdot T(n/b) + f(n)$

IDEA: Compare $n^{\log_b a}$ with $f(n)$.
Master Method — Case I

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

\[ T(n) = \Theta(n^{\log_b a}) \]
Master Method — Case 2

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

ARITHMETICALLY INCREASING

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

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

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

*and \( f(n) \) satisfies the regularity condition that \( a f(n/b) \leq c f(n) \) for some constant \( c < 1 \).
Master–Method Cheat Sheet

Solve

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

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

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

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

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

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

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

- \( T(n) = 4T(n/2) + n^2 / \log n \)
  Master method does not apply!
  Answer is \( T(n) = \Theta(n^2 \log \log n). \) (Prove by substitution.)
CILK LOOPS
Loop Parallelism in Cilk

Example: In-place matrix transpose

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

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

A \rightarrow A^T

// 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;
    }
}

The iterations of a cilk_for loop execute in parallel.
Implementation of Parallel Loops

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

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

```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;
    }
}

void recur(size_t lo, size_t hi) // half open
{
    if (hi > lo + 1) {
        size_t mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid, hi);
        cilk_sync;
        return;
    }
    size_t i = lo;
    for (size_t 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**

**lifted**

loop body
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: \( T_1(n)/T_\infty(n) = \Theta(n) \)
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

Vector addition

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

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

**Span:** $T_\infty = \Theta(lgn)$

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

Includes substantial overhead!
Coarsening Parallel Loops

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

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

```
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

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
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

Work: $T_1 = n \cdot I + \frac{n}{G - 1} \cdot S$

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

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

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
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

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));
}

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:** \( T_1/T_\infty = \Theta(\sqrt{n}) \)
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
# Cilk example

cilk_for (int i=0; i<n; ++i) {
    foo(i);
}
```

**Not this:**

```cilk
# Non-Cilk example

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_for (int i=0; i<2; ++i) {
     for (int j=0; j<n; ++j) {
       f(i,j);
     }
   }
```

   **Not this:**
   ```for (int j=0; j<n; ++j) {
     cilk_for (int i=0; i<2; ++i) {
       f(i,j);
     }
   }
```

   (Cilkview will show a high burdened parallelism.)
SPEED LIMIT
PER ORDER OF 6.172

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_for (int i=1; 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 $\approx (10^3)^2 = 10^6$. 
Recursive Matrix Multiplication

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

**Note:** The dimension $n$ does not enter into the calculation.
Divide-and-Conquer Matrices

\[ M \]

\[ n \]

\[ n_M \]
Divide-and-Conquer Matrices

\[
\begin{array}{cccc}
M_{00} & M_{01} \\
M_{10} & M_{11} \\
\end{array}
\]

\[
\begin{align*}
M_{00} &= M \\
M_{01} &= M + \frac{n}{2} \\
M_{10} &= M + n_M \times \frac{n}{2} \\
M_{11} &= M + (n_M + 1) \times \frac{n}{2} \\
\end{align*}
\]

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

\[
M_{rc} = M + (r \times n_M + c) \times \frac{n}{2}
\]
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 {
        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);
    }
}
```
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);
  }
}
```

The compiler can assume that the input matrices are not aliased.
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)
{
  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);
  }
}
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_add(C, n_C, D, n_D, n);
        free(D);
    }
}
```

Assert for debugging purposes that \( n \) is a power of \( 2 \).
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 {
    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);
  }
}
```

Coarsen the leaves of the recursion to lower the overhead for serial execution.
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*(_##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,0,1), 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,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;
        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];
                }
            }
            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];
            }
        }
        free(D);
    }
}

D&C Matrix Multiplication

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
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);
  }
}
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);
    }
}
```

The temporary array has underlying row size n.
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), n_A, X(B,1,0), n_B, n/2);
  cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,0), n_A, X(B,1,1), n_B, n/2);
  cilk_spawn mm_dac(X(D,1,1), n_D, X(A,1,0), n_A, X(B,1,1), n_B, n/2);
  cilk_sync;
  m_add(C, n_C, D, n_D, n);
  free(D);
}

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 {
        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);
    }
}
```

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

Wait for all spawned subcomputations to complete.
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,0,1), 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);
        #endif
        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); // C += D
        
        // D&C Matrix Multiplication
        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];
            }
        }
        free(D);
    }
}

void m_add (double *restrict C, int n_C,
            double *restrict D, int n_D,
            int n)
{
    // C += D
    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*(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,1,0), n_A, X(B,1,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,0,1), n_A, X(B,0,1), 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);
        cilkSpawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,0,1), n_B, n/2);
        cilkSpawn mm_dac(X(D,0,1), n_D, X(A,1,0), n_A, X(B,1,0), n_B, n/2);
        cilkSpawn mm_dac(X(D,1,0), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        cilkSpawn 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: \( A_\infty(n) = \Theta(\log n) \)
Work of Matrix Multiplication

\[
\begin{align*}
\text{Work:} \quad M_1(n) &= 8M_1(n/2) + A_1(n) + \Theta(1) \\
&= 8M_1(n/2) + \Theta(n^2) \\
&= \Theta(n^3)
\end{align*}
\]

**Case 1**
\[
\begin{align*}
n^{\log_{b} a} &= n^{\log_{2} 8} = n^3 \\
n(n) &= \Theta(n^2)
\end{align*}
\]
Span of Matrix Multiplication

\[
\text{Span: } M_\infty(n) = M_\infty(n/2) + A_\infty(n) + \Theta(1) \\
= M_\infty(n/2) + \Theta(\log n) \\
= \Theta(\log^2 n)
\]

\[n^{\log_b a} = n^{\log_2 1} = 1\]
\[f(n) = \Theta(n^{\log_b a \log_2 1})\]
Parallelism of Matrix Multiply

\[ \text{Work: } M_1(n) = \Theta(n^3) \]
\[ \text{Span: } M_\infty(n) = \Theta(\lg^2 n) \]

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

For 1000 \times 1000 matrices, parallelism \approx (10^3)^3/10^2 = 10^7.
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);
    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);
  }
}
```
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);
    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_sync;
    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;
  }
}

Saves space, but at what expense?
Work 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 + (n*(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_sync;
  }
  cilk_sync;
}
```

**Case 1**

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

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

\[ \text{Work: } M_1(n) = 8M_1(n/2) + \Theta(1) = \Theta(n^3) \]
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),
    cilk_sync);
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1),
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1),
    cilk_spawn 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
Merging Two Sorted Arrays

```c
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--;
    }
}
```

Time to merge n elements $= \Theta(n)$. 

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
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);
    }
}
```

**Case 2**

\[ n^{\log_{b}a} = n^{\log_{2}2} = n \]

\[ f(n) = \Theta(n^{\log_{b}a\log^{0}n}) \]

**Work:**

\[ T_{1}(n) = 2T_{1}(n/2) + \Theta(n) \]

\[ = \Theta(n \log n) \]
Span of Merge Sort

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

\[ n^{\log_b a} = n^{\log_2 1} = 1 \]

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

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
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!
Parallel 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 $(3/4)n$. 

---

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
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, nb-mb);
        cilk_sync;
    }
}
```

Coarsen base cases for efficiency.
```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**

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

\[ f(n) = \Theta(n^{\log_{b}a} \lg^{1}n) \]

**Span:**

\[ T_{\infty}(n) = T_{\infty}(3n/4) + \Theta(\lg n) \]
\[ = \Theta(\lg^{2}n) \]
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, nb-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 \( T_1(k) \leq c_1 k - c_2 \lg k, \) where \( c_1, c_2 > 0. \) Prove that the relation holds, and solve for \( c_1 \) and \( c_2. \)

\[
T_1(n) \leq c_1(\alpha n) - c_2 \lg(\alpha n) + c_1(1-\alpha)n - c_2 \lg((1-\alpha)n) + \Theta(\lg n)
\]
\[
\leq c_1 n - c_2 \lg(\alpha n) - c_2 \lg((1-\alpha)n) + \Theta(\lg n)
\]
\[
\leq c_1 n - c_2 (\lg(\alpha(1-\alpha)) + 2 \lg n ) + \Theta(\lg n)
\]
\[
\leq c_1 n - c_2 \lg n - (c_2(\lg n + \lg(\alpha(1-\alpha)))) - \Theta(\lg n)
\]
\[
\leq c_1 n - c_2 \lg n ,
\]
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:** \( \frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n) \)
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-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-n/2);
        cilk_sync;
        p_merge(B, C, n/2, C+n/2, n-n/2);
    }
}
```

**Case 2**

\[ n^{\log_b a} = n^{\log_2 1} = 1 \]

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

**Span:**

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

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

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

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

**Parallelism:** \( \frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n) \)
SPEED LIMIT
PER ORDER OF 6.172

TABLEAUCONSTRUCTION
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic programming

- Longest common subsequence
- Edit distance
- Time warping

**Work:** $\Theta(n^2)$. 
Recursive Construction

Parallel code

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

Parallel code

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

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
Recursive Construction

Span: \( T_\infty(n) = 3T_\infty(n/2) + \Theta(1) \)

\[ = \Theta(n^{\lg 3}) \]

Parallel code

I;
cilk_spawn II;
III;
cilk_sync;
IV;

Case 1
\( n^{\log_b a} = n^{\log_2 3} = n^{\lg 3} \)
\( f(n) = \Theta(1) \)
Analysis of Tableau Constr.

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

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

---

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

\[
\begin{array}{ccc}
\text{I} & \text{II} & \text{IV} \\
\text{III} & \text{V} & \text{VII} \\
\text{VI} & \text{VIII} & \text{IX}
\end{array}
\]

**Work:**
\[
T_1(n) = 9T_1(n/3) + \Theta(1)
= \Theta(n^2)
\]

**Case 1**
\[
\begin{align*}
n^\log_b a &= n^\log_3 9 = n^2 \\
f(n) &= \Theta(1)
\end{align*}
\]
### A More–Parallel Construction

<table>
<thead>
<tr>
<th>I</th>
<th>II</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>III</td>
<td>V</td>
<td>VII</td>
</tr>
<tr>
<td>VI</td>
<td>VIII</td>
<td>IX</td>
</tr>
</tbody>
</table>

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

$= \Theta(n^{\log_3 5})$

**Case 1**

$n^{\log_b a} = n^{\log_3 5}$

$f(n) = \Theta(1)$
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.