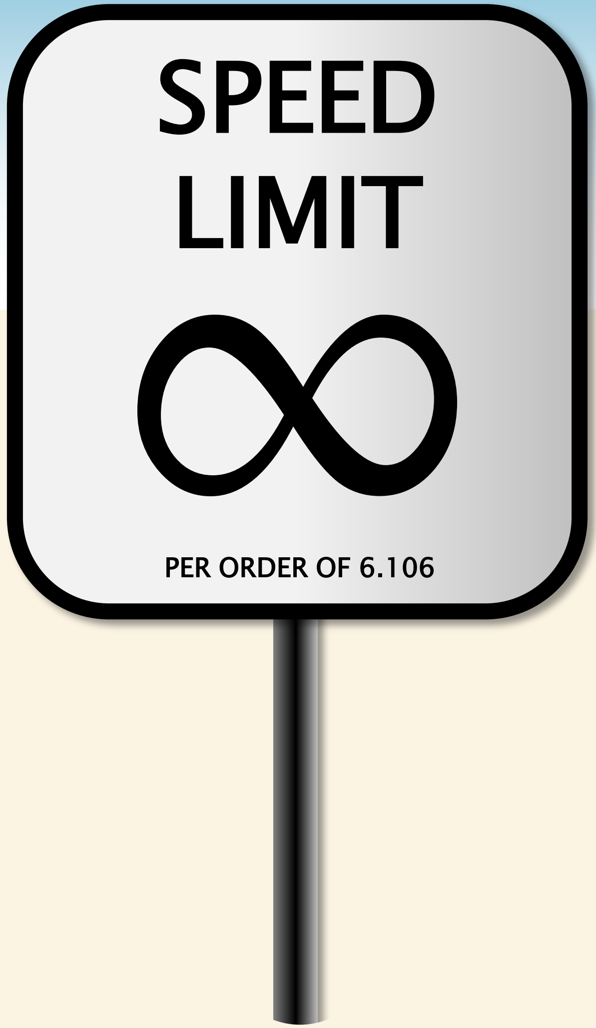
Software
Performance
Engineering



LECTURE 10

Task-Parallel Algorithms II

Charles E. Leiserson

October 13, 2022

Matrix Multiplication

Square-Matrix Multiplication

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

C A B

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

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

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 $\approx (10^3)^2 = 10^6$.

Recursive Matrix Multiplication

Divide and conquer — uses cache more efficiently, as we'll see later in the term.

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

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

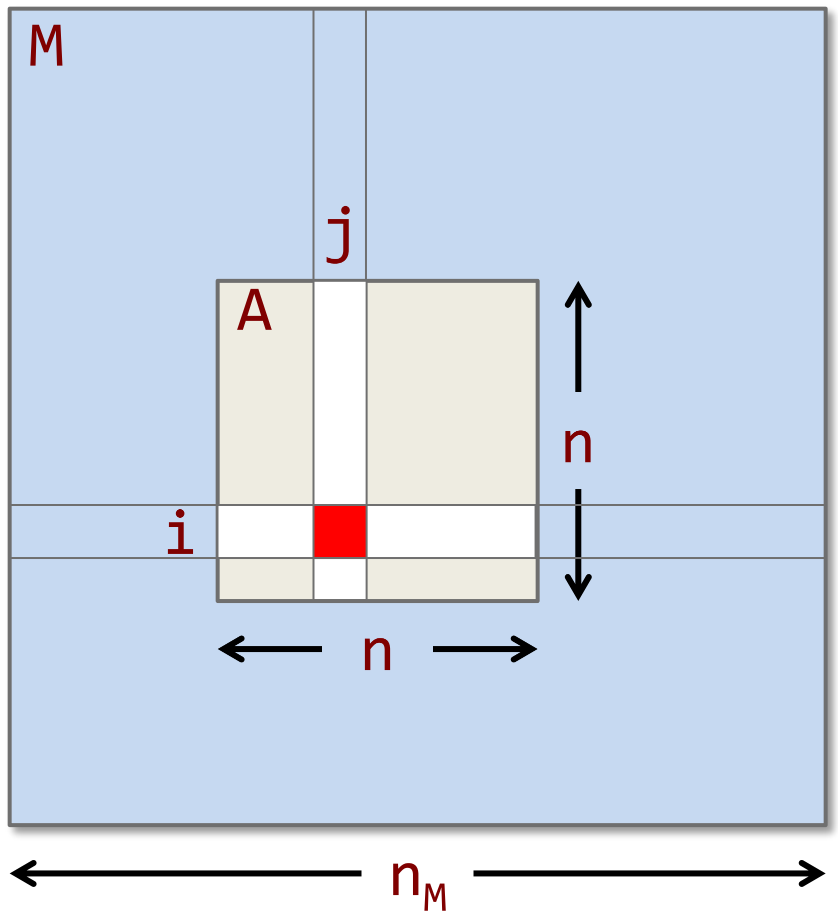
8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.

Representation of Submatrices

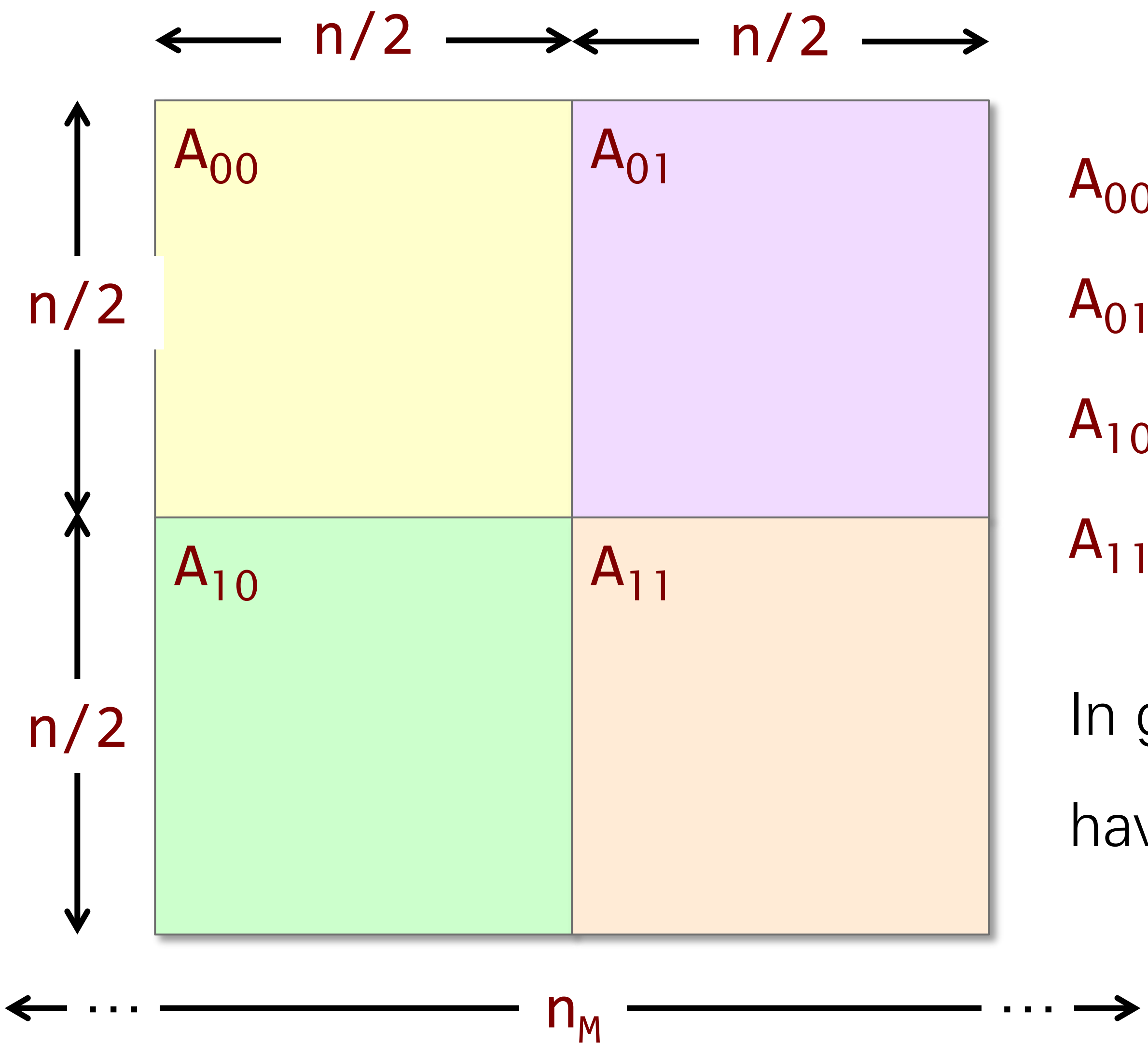
Row-major layout

If 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 $A[n_M i + j]$.

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



$A_{00} = A$

$A_{01} = A + (n/2)$

$A_{10} = A + n_M(n/2)$

$A_{11} = A + (n_M+1)(n/2)$

In general, for $r, c \in \{0,1\}$, we have $A_{rc} = A + (r n_M + c)(n/2)$

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 {
    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_scope {
      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);
} }
```

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_scope {
      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);
} }
```

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 {
    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_scope {
      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);
} }
```

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 {
    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_scope {
      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);
} }
```

The three input matrices are $n \times n$

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_scope {
      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);
} }
```

The function adds the matrix product AB to matrix C

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_scope {
      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);
} }
```

Assert that n is a power of 2

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_scope {
      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);
} }
```

Coarsen the leaves of the recursion to lower the overhead.

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_scope {
      cilk_spawn mm_dac(
      cilk_spawn mm_dac(
      cilk_spawn mm_dac(
      cilk_spawn mm_dac(
      cilk_spawn mm_dac(
      cilk_spawn mm_dac(
      cilk_spawn mm_dac(
      cilk_spawn mm_dac(
    }
    m_add(C, n_C, D, n_D
    free(D);
} }
```

Coarsen the leaves of the recursion to lower the overhead.

```
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 {
    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_scope {
      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);
} }
```

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 * si
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_scope {
      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);
} }
```

The temporary array D has underlying row size 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 {
    double *D = malloc(n * n           zeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_scope {
      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,  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    ), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1     A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac(X(D,0,0), n_D, X(A                       , n/2);
      cilk_spawn mm_dac(X(D,0,1), n_D, X(                        , n/2);
      cilk_spawn mm_dac(X(D,1,0), n_D, X(                        , n/2);
      cilk_spawn mm_dac(X(D,1,1), n_D, X(                        , n/2);
    }
    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

```
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_scope {
      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);
} }
```

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)*[illegible])
    cilk_scope {
      cilk_spawn mm_dac(X(C,0,0)[illegible], X(A,0,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac(X(C,0[illegible]n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac(X(C[illegible]0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac[illegible],1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm[illegible](X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn[illegible]_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
      cilk_sp[illegible]wn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
      ci[illegible]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.

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_scope {
      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    ,0), 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_C    (A,1,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac(X(D,0,0), n      X(A,0,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac(X(D,0,1),     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);
} }
```

Add the temporary matrix D into the output matrix C

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_scope {
      cilk_spawn mm_dac(X(C,0,
      cilk_spawn mm_dac(X(C,0,
      cilk_spawn mm_dac(X(C,1,
      cilk_spawn mm_dac(X(C,1,
      cilk_spawn mm_dac(X(D,0,
      cilk_spawn mm_dac(X(D,0,
      cilk_spawn mm_dac(X(D,1,
      cilk_spawn mm_dac(X(D,1,
    }
    m_add(C, n_C, D, n_D, n);
    free(D);
} }
```

Add the temporary matrix D into the output matrix 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];
    }
  }
}
```

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_scope {
      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);
} }
```

Clean up, and then return.

Analysis of Divide-and-Conquer Matrix Multiplication

Master-Method Cheat Sheet

Solve

$$\boxed{T(n) = a\,T(n/b) + f(n)\ ,}$$

where $a \geqslant 1$ 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)
$\Rightarrow T(n) = \Theta(f(n))$.

https://tinyurl.com/mm-cheat

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_scope {
      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,
      cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A,
      cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A,
      cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A,
      cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A,
      cilk_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A,
    }
    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

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

maximum

CASE 2

$n^{\log_b a} = n^{\log_2 1} = 1$

$f(n) = \Theta(n^{\log_b a} \lg^1 n)$

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

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

$= \Theta(\lg^2 n)$

Parallelism of Matrix Multiply

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

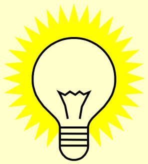
Span: $M_\infty(n) = \Theta(\lg^2 n)$

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

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

Temporaries

```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_scope {
      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);
```



IDEA

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

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

How to Avoid the Temporary?

```
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_scope {
      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);
} }
```

No-Temp Matrix Multiplication

```
void mm_dac2(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_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    }
    cilk_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
} } }
```

Do 4 subproblems in parallel…

…and when they're done, do the other 4.

No-Temp Matrix Multiplication

```
void mm_dac2(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_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    }
    cilk_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
} } }
```

Reuse C without racing.

Saves space, but at what expense?

Work of No-Temp Multiply

```c
void mm_dac2(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_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    }
    cilk_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,1), n_A, X(B
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,1), n_A, X(B
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,1), n_A, X(B
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,1), n_A, X(B
} } }
```

CASE 1

$n^{\log_b a} = n^{\log_2 8} = n^3$

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

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

$= \Theta(n^3)$

Span of No-Temp Multiply

```
void mm_dac2(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_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    }
    cilk_scope {
      cilk_spawn mm_dac2(X(C,0,0), n_C, X(A,0,1), n_A,
      cilk_spawn mm_dac2(X(C,0,1), n_C, X(A,0,1), n_A,
      cilk_spawn mm_dac2(X(C,1,0), n_C, X(A,1,1), n_A,
      cilk_spawn mm_dac2(X(C,1,1), n_C, X(A,1,1), n_A,
} } }
```

max

max

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

Faster in practice!

Parallel Merge Sort

Merge Sort

```c
void merge_sort(int *restrict A,       // unsorted input array of length n
                int n,                 // # elements in A (and B)
                int *restrict B)       // sorted output array of length n
{
  assert(n > 0);                       // check that # elements is positive
  if (n == 1) {                        // should coarsen the recursion
    B[0] = A[0]; return;               // 1-element array is sorted
  }
  int C[n];                            // create a temporary array C
  merge_sort(A, n/2, C);               // sort the lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);     // sort the upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);      // merge the two halves into B
}
```

- Classic recursive algorithm for sorting.
- Not in place: requires auxiliary array.
- Asymptotically optimal running time for a comparison sort: $\Theta(n \lg n)$.

Merge Sort

```
void merge_sort(int *restrict A,     // unsorted input array of length n
                int n,               // # elements in A (and B)
                int *restrict B)     // sorted output array of length n
{
  assert(n > 0);                     // check that # elements is positive
  if (n == 1) {                      // should coarsen the recursion
    B[0] = A[0]; return;             // 1-element array is sorted
  }
  int C[n];                          // create a temporary array C
  merge_sort(A, n/2, C);             // sort the lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);   // sort the upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);    // merge the two halves into B
}
```

Merge Sort

```c
void merge_sort(int *restrict A,     // unsorted input array of length n
                int n,               // # elements in A (and B)
                int *restrict B)     // sorted output array of length n
{
  assert(n > 0);                     // check that # elements is positive
  if (n == 1) {                      // should coarsen the recursion
    B[0] = A[0]; return;             // 1-element array is sorted
  }
  int C[n];                          // create a temporary array C
  merge_sort(A, n/2, C);             // sort the lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);   // sort the upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);    // merge the two halves into B
}
```

Merge Sort

```c
void merge_sort(int *restrict A,      // unsorted input array of length n
                int n,                // # elements in A (and B)
                int *restrict B)      // sorted output array of length n
{
  assert(n > 0);                      // check that # elements is positive
  if (n == 1) {                       // should coarsen the recursion
    B[0] = A[0]; return;              // 1-element array is sorted
  }
  int C[n];                           // create a temporary array C
  merge_sort(A, n/2, C);              // sort the lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);    // sort the upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);     // merge the two halves into B
}
```

Merge Sort

```c
void merge_sort(int *restrict A,      // unsorted input array of length n
                int n,                // # elements in A (and B)
                int *restrict B)      // sorted output array of length n
{
  assert(n > 0);                      // check that # elements is positive
  if (n == 1) {                       // should coarsen the recursion
    B[0] = A[0]; return;              // 1-element array is sorted
  }
  int C[n];                           // create a temporary array C
  merge_sort(A, n/2, C);              // sort the lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);    // sort the upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);     // merge the two halves into B
}
```

Merge Sort

```c
void merge_sort(int *restrict A,        // unsorted input array of length n
                int n,                  // # elements in A (and B)
                int *restrict B)        // sorted output array of length n
{
  assert(n > 0);                        // check that # elements is positive
  if (n == 1) {                         // should coarsen the recursion
    B[0] = A[0]; return;                // 1-element array is sorted
  }
  int C[n];                             // create a temporary array C
  merge_sort(A, n/2, C);                // sort the lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);      // sort the upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);       // merge the two halves into B
}
```

Merge Sort

```
void merge_sort(int *restrict A,      // unsorted input array of length n
                int n,                // # elements in A (and B)
                int *restrict B)      // sorted output array of length n
{
  assert(n > 0);                      // check that # elements is positive
  if (n == 1) {                       // should coarsen the recursion
    B[0] = A[0]; return;              // 1-element array is sorted
  }
  int C[n];                           // create a temporary array C
  merge_sort(A, n/2, C);              // sort the lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);    // sort the upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);     // merge the two halves into B
}
```

Merge Sort

```
void merge_sort(int *restrict A,       // unsorted input array of length n
                int n,                 // # elements in A (and B)
                int *restrict B)       // sorted output array of length n
{
  assert(n > 0);                       // check that # elements is positive
  if (n == 1) {                        // should coarsen the recursion
    B[0] = A[0]; return;               // 1-element array is sorted
  }
  int C[n];                            // create a temporary array C
  merge_sort(A, n/2, C);               // sort the lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);     // sort the upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);      // merge the two halves into B
}
```

Merging Two Sorted Arrays

```c
void merge(int *restrict A, int na,
           int *restrict B, int nb,
           int *restrict C) {
  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 = $\Theta(n)$

C								

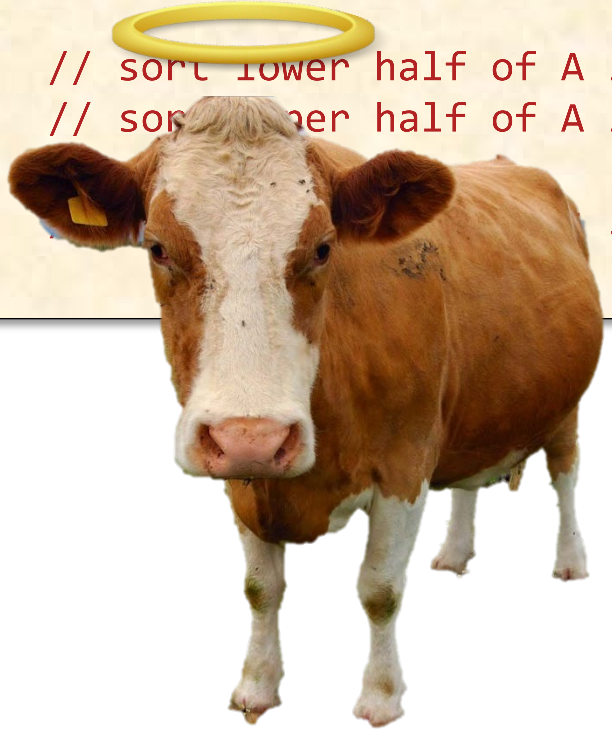
A	3	12	19	46
B	4	14	21	23

How to Parallelize?

```
void merge_sort(int *restrict A,          // unsorted input array
                int n,                    // # elements in A (and B)
                int *restrict B)          // sorted output array
{
  assert(n > 0);                          // check # elements is positive
  if (n == 1) {                           // should coarsen recursion
    B[0] = A[0]; return;                  // 1-element array is sorted
  }
  int C[n];                               // create a temporary array C
                                          // sort lower half of A into C
  merge_sort(A+n/2, n-n/2, C+n/2);        // sort upper half of A into C
  merge(C, n/2, C+n/2, n-n/2, B);         // merge the two halves into B
}
```

Parallel Merge Sort

```
void p_merge_sort(int *restrict A,        // unsorted input array
                  int n,                   // # elements in A (and B)
                  int *restrict B)         // sorted output array
{
  assert(n > 0); // check # elements is positive
  if (n == 1) {  // should coarsen recursion
    B[0] = A[0]; return; // 1-element array is sorted
  }
  int C[n];      // create a temporary array C
  cilk_scope {
    cilk_spawn p_merge_sort(A, n/2, C);  // sort lower half of A into C
    p_merge_sort(A+n/2, n-n/2, C+n/2);   // so   per half of A into C
  }
  merge(C, n/2, C+n/2, n-n/2, B);                       into B
}
```



HOLY COW!
That was easy!

Parallel Merge Sort

```
void p_merge_sort(int *restrict A,        // unsorted input array
                  int n,                  // # elements in A (and B)
                  int *restrict B)        // sorted output array
{
  assert(n > 0); // check # elements is positive
  if (n == 1) {  // should coarsen recursion
    B[0] = A[0]; return; // 1-element array is sorted
  }
  int C[n];      // create a temporary array C
  cilk_scope {
    cilk_spawn p_merge_sort(A, n/2, C);  // sort lower half of A into C
    p_merge_sort(A+n/2, n-n/2, C+n/2);   // sort upper half of A into C
  }
  merge(C, n/2, C+n/2, n-n/2, B);        // merge the two halves into B
}
```

Parallel Merge Sort Animation

```
void p_merge_sort(int *restrict A,       // unsorted input array
                  int n,                 // # elements in A (and B)
                  int *restrict B)       // sorted output array
{
  ⋮
cilk_scope {
    cilk_spawn p_merge_sort(A, n/2, C);  // sort lower half of A into C
    p_merge_sort(A+n/2, n-n/2, C+n/2);   // sort upper half of A into C
  }
  merge(C, n/2, C+n/2, n-n/2, B);        // merge the two halves into B
}
```

19	3	12	46	33	4	21	14

Parallel Merge Sort Animation

```
void p_merge_sort(int *restrict A,        // unsorted input array
                  int n,                  // # elements in A (and B)
                  int *restrict B)        // sorted output array
{
  ⋮
cilk_scope {
    cilk_spawn p_merge_sort(A, n/2, C);  // sort lower half of A into C
    p_merge_sort(A+n/2, n-n/2, C+n/2);   // sort upper half of A into C
  }
  merge(C, n/2, C+n/2, n-n/2, B);        // merge the two halves into B
}
```

19	3	12	46	33	4	21	14

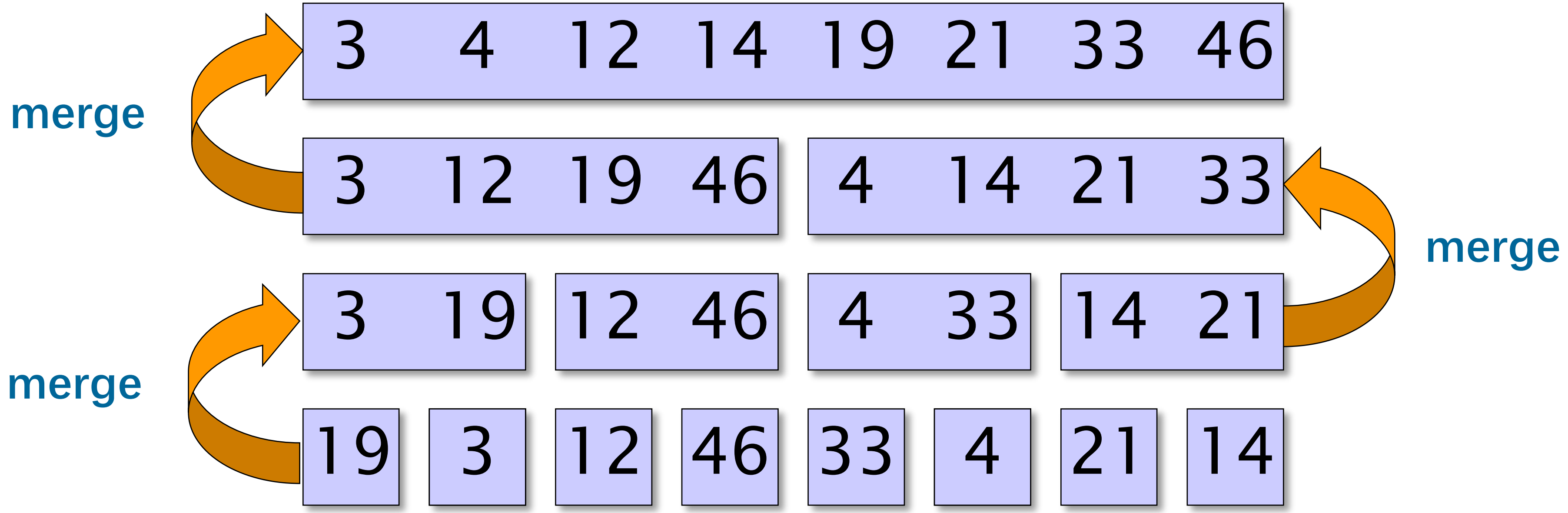
Parallel Merge Sort Animation

```
void p_merge_sort(int *restrict A,         // unsorted input array
                  int n,                   // # elements in A (and B)
                  int *restrict B)         // sorted output array
{
  ⋮
cilk_scope {
    cilk_spawn p_merge_sort(A, n/2, C);  // sort lower half of A into C
    p_merge_sort(A+n/2, n-n/2, C+n/2);   // sort upper half of A into C
  }
  merge(C, n/2, C+n/2, n-n/2, B);        // merge the two halves into B
}
```

19	3	12	46	33	4	21	14

Parallel Merge Sort Animation

```
void p_merge_sort(int *restrict A,         // unsorted input array
                  int n,                   // # elements in A (and B)
                  int *restrict B)         // sorted output array
{
  ⋮
cilk_scope {
    cilk_spawn p_merge_sort(A, n/2, C);   // sort lower half of A into C
    p_merge_sort(A+n/2, n-n/2, C+n/2);    // sort upper half of A into C
  }
  merge(C, n/2, C+n/2, n-n/2, B);         // merge the two halves into B
}
```



Work of Parallel Merge Sort

```
void p_merge_sort(int *restrict A,        // unsorted input array
                  int n,                  // # elements in A (and B)
                  int *restrict B)        // sorted output array
{
  assert(n > 0); // check # elements is positive
  if (n == 1) {  // should coarsen recursion
    B[0] = A[0]; return; // 1-element array is sorted
  }
  int C[n];      // create a temporary array C
  cilk_scope {
    cilk_spawn p_merge_sort(A, n/2, C);  // sort lower half of A into C
    p_merge_sort(A+n/2, n-n/2, C+n/2);   // sort upper half of A into C
  }
  merge(C, n/2, C+n/2, n-n/2, B);        // merge the two halves into B
}
```

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

$= \Theta(n \lg n)$

CASE 2

$n^{\log_b a} = n^{\log_2 2} = n$

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

Span of Parallel Merge Sort

```
void p_merge_sort(int *restrict A,        // unsorted input array
                  int n,                  // # elements in A (and B)
                  int *restrict B)        // sorted output array
{
  assert(n > 0); // check # elements is positive
  if (n == 1) {  // should coarsen recursion
    B[0] = A[0]; return; // 1-element array is sorted
  }
  int C[n];      // create a temporary array C
  cilk_scope {
    cilk_spawn p_merge_sort(A, n/2, C);  // sort lower half of A into C
    p_merge_sort(A+n/2, n-n/2, C+n/2);   // sort upper half of A into C
  }
  merge(C, n/2, C+n/2, n-n/2, B);        // merge the two halves into B
}
```

max

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: $T_1(n) = \Theta(n \lg n)$

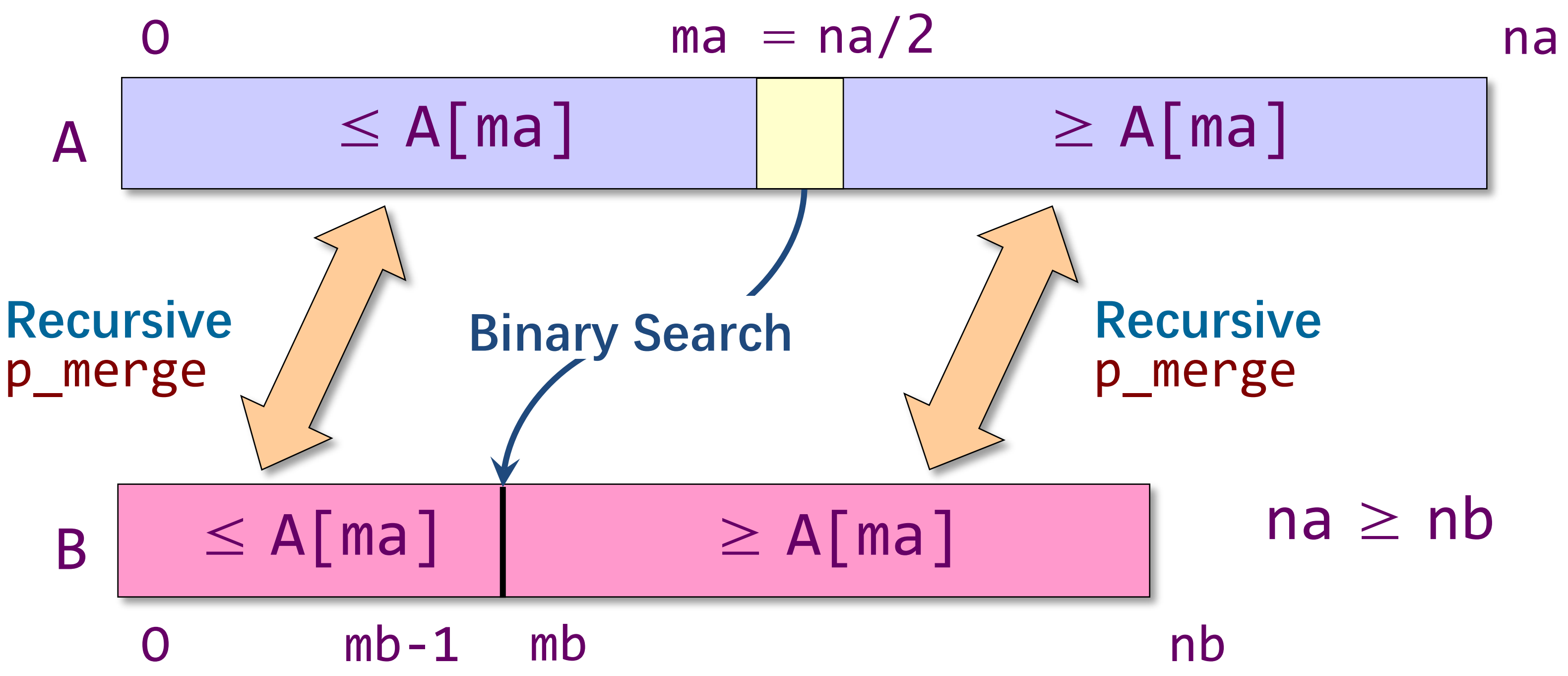
Span: $T_\infty(n) = \Theta(n)$

puny

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$

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 `3n/4`.

Parallel Merge

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

Parallel Merge

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

Parallel Merge

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

Parallel Merge

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

Parallel Merge

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

Work of Parallel Merge

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

Is parallel merge asymptotically work efficient?

Work Efficiency

Definition

- Let $T_S(n)$ be the running time of the best serial algorithm on a given input of size n.
- Let $T_1(n)$ be the work of a parallel program (its running time on 1 processor) on the same input.
- The **work overhead** of the parallel program is the worst-case ratio $\lambda(n) = T_1(n)/T_S(n)$.
- We say that the parallel program is **work efficient** if $T_1(n) \approx T_S(n)$ and **asymptotically work efficient** if $T_1 = \Theta(T_S)$.

Work-First Principle

Suppose that the worst-case work overhead for a parallel algorithm on a given input is $\lambda = T_1/T_S$.

The **WORK LAW** says that

$$\begin{aligned} T_P &\geqslant T_1/P \\ &= \lambda \cdot T_S/P . \end{aligned}$$

Lessons

- If λ is large, we cannot get near-perfect linear speedup over the good serial code no matter how many processors we run on.
- We waste processing power proportional to the work overhead.
- We must minimize work *first*, ahead of maximizing parallelism, if we want efficiency.

Work of Parallel Merge

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

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

Analysis of Work Recurrence

$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,
where $1/4 \leqslant \alpha \leqslant 3/4$.

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

$$
\begin{aligned}
T_1(n) &\leqslant c_1(\alpha n) - c_2\lg(\alpha n) + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&= c_1 n - c_2\lg(\alpha n) - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&= c_1 n - c_2\,(\lg(\alpha(1-\alpha)) + 2\lg n\,) + \Theta(\lg n) \\
&= c_1 n - c_2\lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\leqslant c_1 n - c_2\lg n
\end{aligned}
$$

if we choose c_2 large enough for sufficiently large n. We then choose c_1 large enough to handle the base cases. Hence, we have $T_1(n) = O(n)$, and since $T_1(n) = \Omega(n)$ trivially, it follows that $T_1(n) = \Theta(n)$.

Span of Parallel Merge

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

CASE 2

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

$f(n) = \Theta(n^{\log_b a} \lg^1 n)$

Span: $T_\infty(n) \leqslant T_\infty(3n/4) + \Theta(\lg n)$

$= \Theta(\lg^2 n)$

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 (Version 2)

```
void p_merge_sort2(int *restrict A,        // unsorted input array
                int n,                     // # elements in A (and B)
                int *restrict B)           // sorted output array
{
  assert(n > 0); // check # elements is positive
  if (n == 1) {  // should coarsen recursion
    B[0] = A[0]; return; // 1-element array is sorted
  }
  int C[n];      // create a temporary array C
  cilk_spawn p_merge_sort2(A, n/2, C);     // sort lower half of A into C
  p_merge_sort2(A+n/2, n-n/2, C+n/2);      // sort upper half of A into C
  cilk_sync;
  p_merge(C, n/2, C+n/2, n-n/2, B);        // merge the two halves into B
}
```

Work of Parallel Merge Sort

```
void p_merge_sort2(int *restrict A,        // unsorted input array
                   int n,                  // # elements in A (and B)
                   int *restrict B)        // sorted output array
{
  assert(n > 0); // check # elements is positive
  if (n == 1) {  // should coarsen recursion
    B[0] = A[0]; return; // 1-element array is sorted
  }
  int C[n];      // create a temporary arra
  cilk_spawn p_merge_sort2(A, n/2, C);     //
  p_merge_sort2(A+n/2, n-n/2, C+n/2);      //
  cilk_sync;
  p_merge(C, n/2, C+n/2, n-n/2, B);        //
}
```

CASE 2

$n^{\log_b a} = n^{\log_2 2} = n$

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

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

$= \Theta(n \lg n)$

Span of Parallel Merge Sort

```
void p_merge_sort2(int *restrict A,        // unsorted input array
                int n,                     // # elements in A (and B)
                int *restrict B)           // sorted output array
{
  assert(n > 0); // check # elements is positive
  if (n == 1) {  // should coarsen recursion
    B[0] = A[0]; return; // 1-element array is sorted
  }
  int C[n];      // create a temporary arra
  cilk_spawn p_merge_sort2(A, n/2, C);    //
  p_merge_sort2(A+n/2, n-n/2, C+n/2);     //
  cilk_sync;
  p_merge(C, n/2, C+n/2, n-n/2, B);       //
}
```

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 Parallel Merge Sort

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

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

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$$