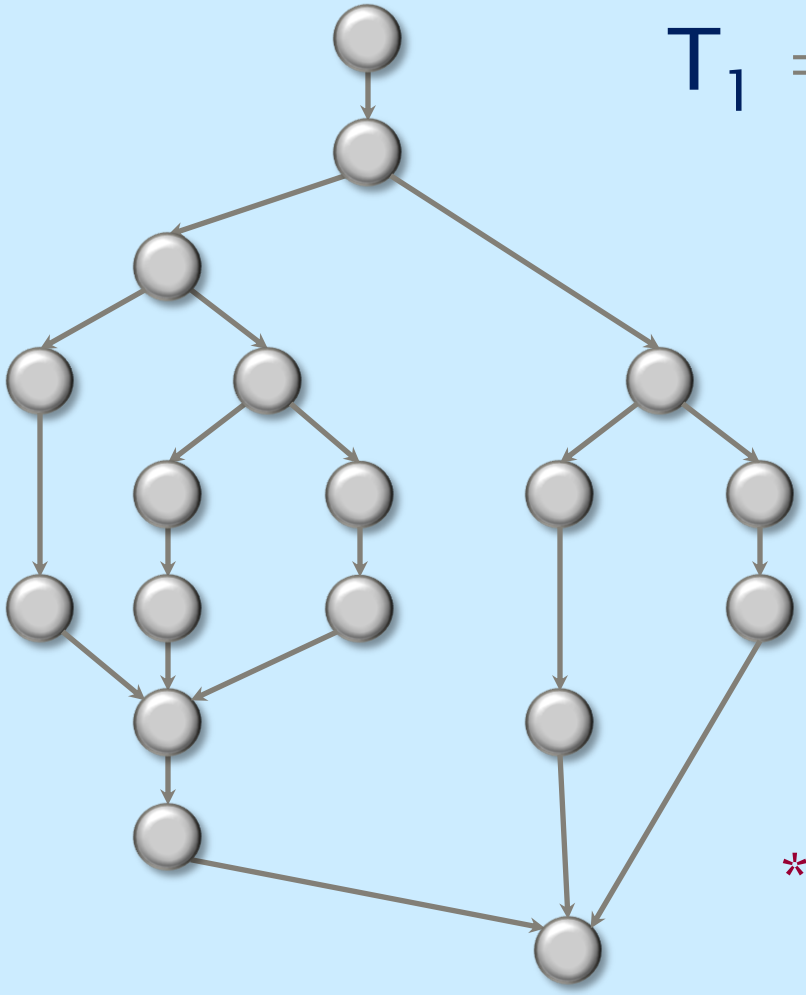
CS 140 : Numerical Examples on Shared Memory with Cilk++

- Matrix-matrix multiplication
- Matrix-vector multiplication
- Hyperobjects

Thanks to Charles E. Leiserson for some of these slides

Work and Span (Recap)

T_P = execution time on P processors



T_1 = *work* T_∞ = *span**

Speedup on p processors
- T_1 / T_p

Parallelism
- T_1 / T_∞

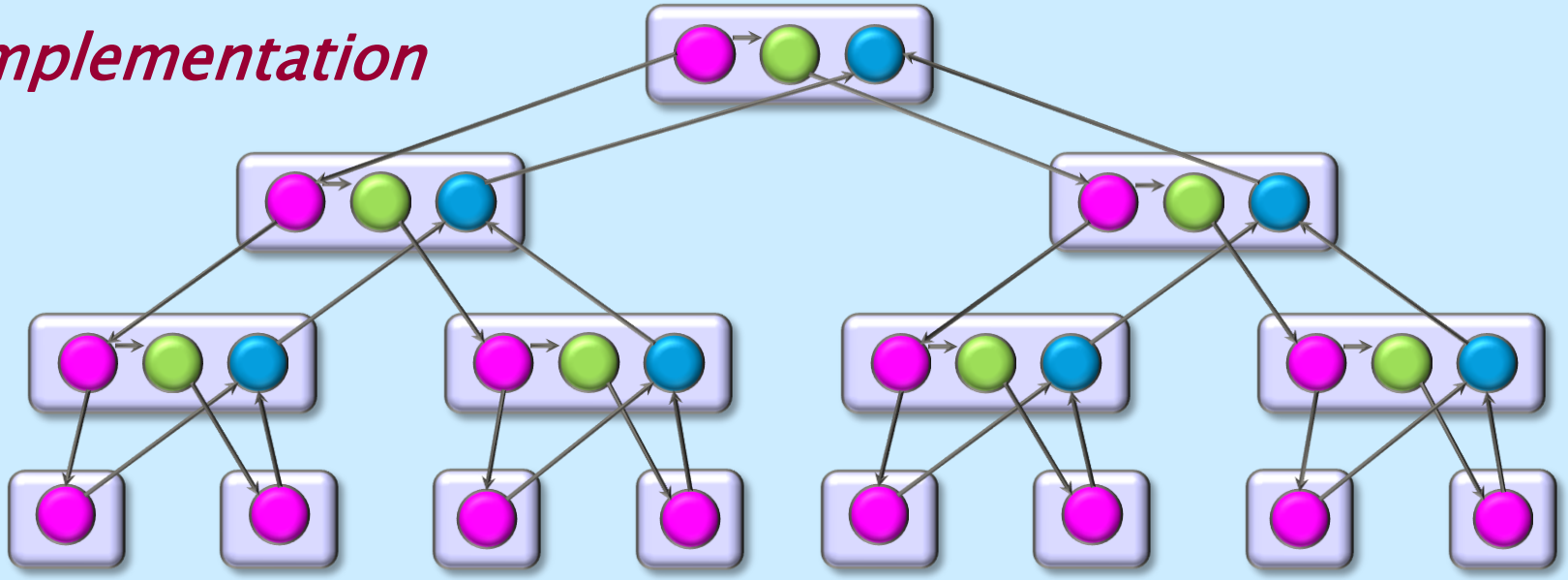
*Also called *critical-path length* or *computational depth*.

Cilk Loops: Divide and Conquer

Vector addition

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

Implementation



G

grain size

Assume that $G = \Theta(1)$.

Work: $T_1 = \Theta(n)$

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

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

Square-Matrix Multiplication

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

C A B

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

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

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

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

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

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

Recursive Matrix Multiplication

Divide and conquer —

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

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

D&C Matrix Multiplication

```
template <typename T>
void MMult(T *C, T *A, T *B, int n) {
  T * D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2);
  cilk_spawn MMult(C12, A11, B12, n/2);
  cilk_spawn MMult(C22, A21, B12, n/2);
  cilk_spawn MMult(C21, A21, B11, n/2);
  cilk_spawn MMult(D11, A11, B21, n/2);
  cilk_spawn MMult(D12, A12, B22, n/2);
  cilk_spawn MMult(D22, A22, B21, n/2);
             MMult(D21, A12, B22, n/2);
  cilk_sync;
  MAdd(C, D, n); // C += D;
}
```

Coarsen for efficiency

Row/column length of matrices

Determine submatrices by index calculation

Matrix Addition

```
template <typename T>
void MMult(T *C, T *A, T *B, int n) {
  T * D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2);
  cilk_spawn MMult(C12, A11, B12, n/2);
  cilk_spawn MMult(C22, A21, B12, n/2);
  cilk_spawn MMult(C21, A21, B11, n/2);
  cilk_sp
  cilk_sp
  cilk_sp

  cilk_sy
  MAdd(C,
}
```

```
template <typename T>
void MAdd(T *C, T *D, int n) {
  cilk_for (int i=0; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
      C[n*i+j] += D[n*i+j];
    }
  }
}
```

Analysis of Matrix Addition

```
template <typename T>
void MAdd(T *C, T *D, int n) {
  cilk_for (int i=0; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
      C[n*i+j] += D[n*i+j];
    }
  }
}
```

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

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

Nested cilk_for statements have the same $\Theta(\lg n)$ *span*

Work of Matrix Multiplication

```
template <typename T>
void MMult(T *C, T *A, T *B, int n) {
 T * D = new T [n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11
  cilk_spawn MMult(C12
   ⋮
  cilk_spawn MMult(D22
             MMult(D21,
  cilk_sync;
  MAdd(C, D, n); // C += 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

```
template <typename T>
void MMult(T *C, T *A, T *B, int n) {
 T * D = new T [n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2);
  cilk_spawn MMult(C12, A11, B12, n/2);
  ⋮
  cilk_spawn MMult(D21, A21, B11, n/2);
             MMult(D22, A21, B12, n/2);
  cilk_sync;
  MAdd(C, D, n, size); // C = 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: $$\frac{M_1(n)}{M_\infty(n)} = \Theta(n^3/\lg^2 n)$$

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

Stack Temporaries

```
template <typename T>
void MMult(T *C, T *A, T *B, int n) {
  T * D = new T [n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2);
  cilk_spawn MMult(C12, A11, B12, n/2);
  cilk_spawn MMult(C22, A21, B12, n/2);
  cilk_spawn MMult(C21, A21, B11, n/2);
  cilk_spawn MMult(D11, A12, B21, n/2);
  cilk_spawn MMult(D12, A12, B22, n/2);
  cilk_spawn MMult(D22, A22, B22, n/2);
             MMult(D21, A22, B21, n/2);
  cilk_sync;
  MAdd(C, D, n); // C += D;
}
```

IDEA: Since minimizing storage tends to yield higher performance, trade off parallelism for less storage.

No-Temp Matrix Multiplication

```
// C += A*B;
template <typename T>
void MMult2(T *C, T *A, T *B, int n) {
  //base case & partition matrices
  cilk_spawn MMult2(C11, A11, B11, n/2);
  cilk_spawn MMult2(C12, A11, B12, n/2);
  cilk_spawn MMult2(C22, A21, B12, n/2);
             MMult2(C21, A21, B11, n/2);
  cilk_sync;
  cilk_spawn MMult2(C11, A12, B21, n/2);
  cilk_spawn MMult2(C12, A12, B22, n/2);
  cilk_spawn MMult2(C22, A22, B22, n/2);
             MMult2(C21, A22, B21, n/2);
  cilk_sync;
}
```

Saves space, but at what expense?

Work of No-Temp Multiply

```
// C += A*B;
template <typename T>
void MMult2(T *C, T *A, T *B, int n) {
  //base case & partition matrices
  cilk_spawn MMult2(C11, A11, B11, n/2);
  cilk_spawn MMult2(C12, A11, B12, n/2);
  cilk_spawn MMult2(C2
             MMult2(C2
  cilk_sync;
  cilk_spawn MMult2(C1
  cilk_spawn MMult2(C1
  cilk_spawn MMult2(C2
             MMult2(C21, A
  cilk_sync;
}
```

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

```
// C += A*B;
template <typename T>
void MMult2(T *C, T *A, T *B, int n) {
  //base case & partition matrices
  cilk_spawn MMult2(C11, A11, B11, n/2);
  cilk_spawn MMult2(C12, A11, B12, n/2);
  cilk_spawn MMult2(C2
             MMult2(C2
  cilk_sync;
  cilk_spawn MMult2(C1
  cilk_spawn MMult2(C1
  cilk_spawn MMult2(C2
             MMult2(C21, A
  cilk_sync;
}
```

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

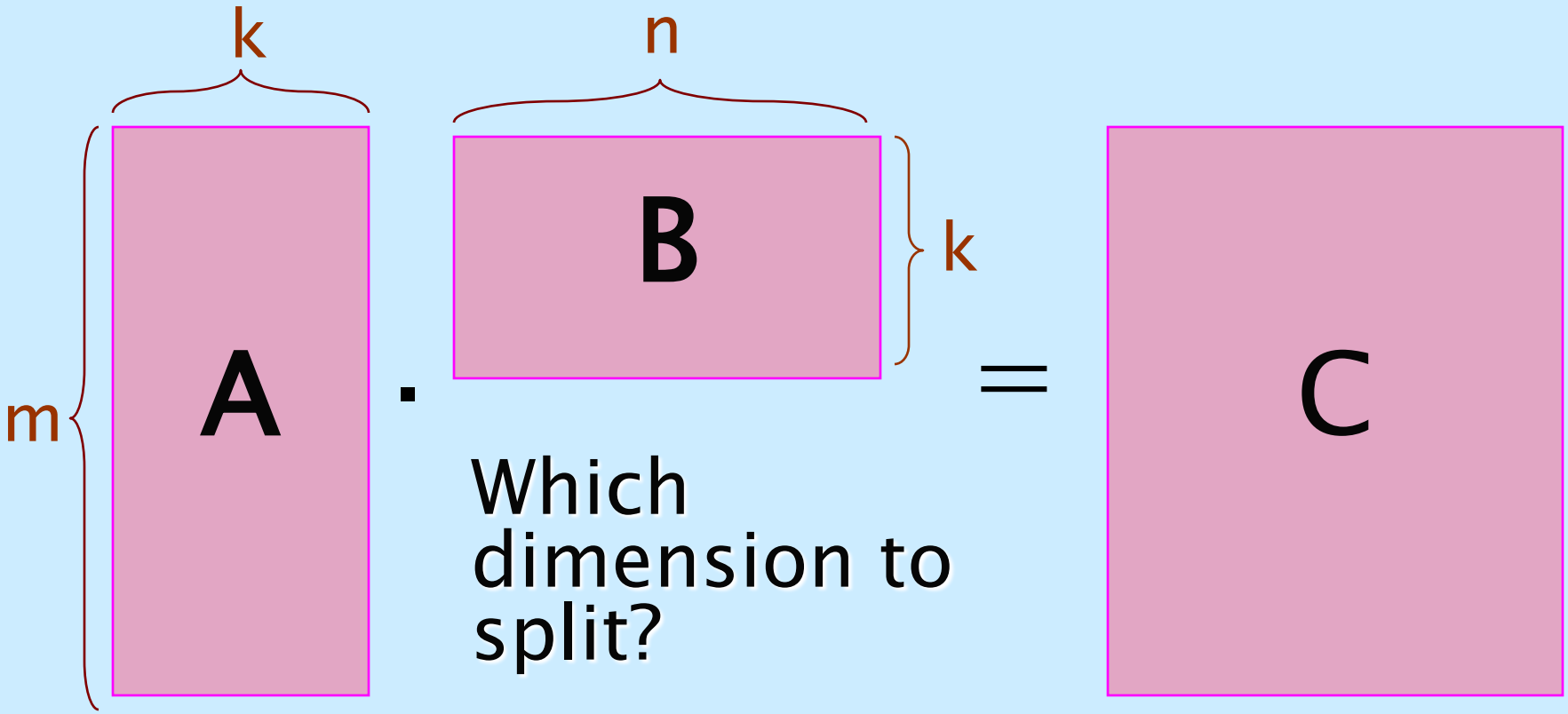
Parallelism: $$\frac{M_1(n)}{M_\infty(n)} = \Theta(n^2)$$

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

Faster in practice!

How general that was?

- Matrices are often rectangular
- Even when they are square, the dimensions are hardly a power of two



General Matrix Multiplication

```
template <typename T>
void MMult3(T *A, T* B, T* C,  int i0, int i1, int j0, int j1, int k0, int k1)
{
    int di = i1 - i0;  int dj = j1 - j0;  int dk = k1 -
    if (di >= dj && di >= dk && di >= THRESHOL
            int mi = i0 + di / 2;
            MMult3 (A, B, C, i0, mi, j0, j1, k0, k1);
            MMult3 (A, B, C, mi, i1, j0, j1, k0, k1);
    } else if (dj >= dk && dj >= THRESHOLD) {
            int mj = j0 + dj / 2;
            MMult3 (A, B, C, i0, i1, j0, mj, k0, k1);
            MMult3 (A, B, C, i0, i1, mj, j1, k0, k1);
    } else if (dk >= THRESHOLD) {
            int mk = k0 + dk / 2;
            MMult3 (A, B, C, i0, i1, j0, j1, k0, mk);
            MMult3 (A, B, C, i0, i1, j0, j1, mk, k1);
    } else { // Iterative (triple-nested loop) multiply }
}
```

Split m if it is the largest

Split n if it is the largest

Split k if it is the largest

```
for (int i = i0; i < i1; ++i) {
  for (int j = j0; j < j1; ++j) {
    for (int k = k0; k < k1; ++k)
    C[i][j] += A[i][k] * B[k][j];
```

Parallelizing General MMult

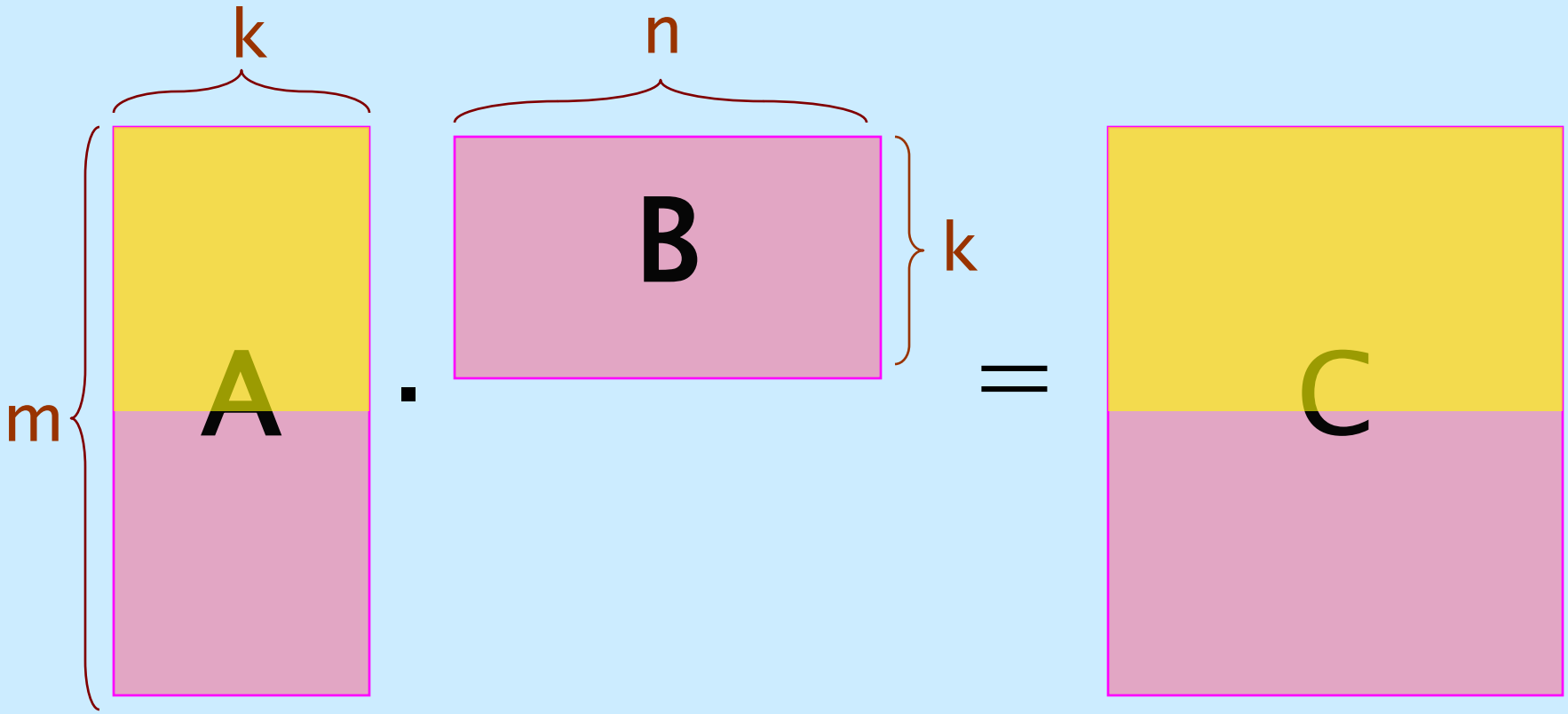
```
template <typename T>
void MMult3(T *A, T* B, T* C,  int i0, int i1, int j0, int j1, int k0, int k1)
{
    int di = i1 - i0;  int dj = j1 - j0;  int dk = k1 - k0;
    if (di >= dj && di >= dk && di >= THRESHOLD) {
            int mi = i0 + di / 2;
            cilk_spawn MMult3 (A, B, C, i0, mi, j0, j1, k0, k1);
            MMult3 (A, B, C, mi, i1, j0, j1, k0, k1);
    } else if (dj >= dk && dj >= THRESHOLD) {
            int mj = j0 + dj / 2;
            cilk_spawn MMult3 (A, B, C, i0, i1, j0, mj, k
            MMult3 (A, B, C, i0, i1, mj, j1, k0, k1);
    } else if (dk >= THRESHOLD) {
            int mk = k0 + dk / 2;
            MMult3 (A, B, C, i0, i1, j0, j1, k0, mk);
            MMult3 (A, B, C, i0, i1, j0, j1, mk, k1);
    } else { // Iterative (triple-nested loop) multiply }
}
```

Unsafe to spawn here unless we use a temporary !

```
for (int i = i0; i < i1; ++i) {
  for (int j = j0; j < j1; ++j) {
    for (int k = k0; k < k1; ++k)
    C[i][j] += A[i][k] * B[k][j];
```

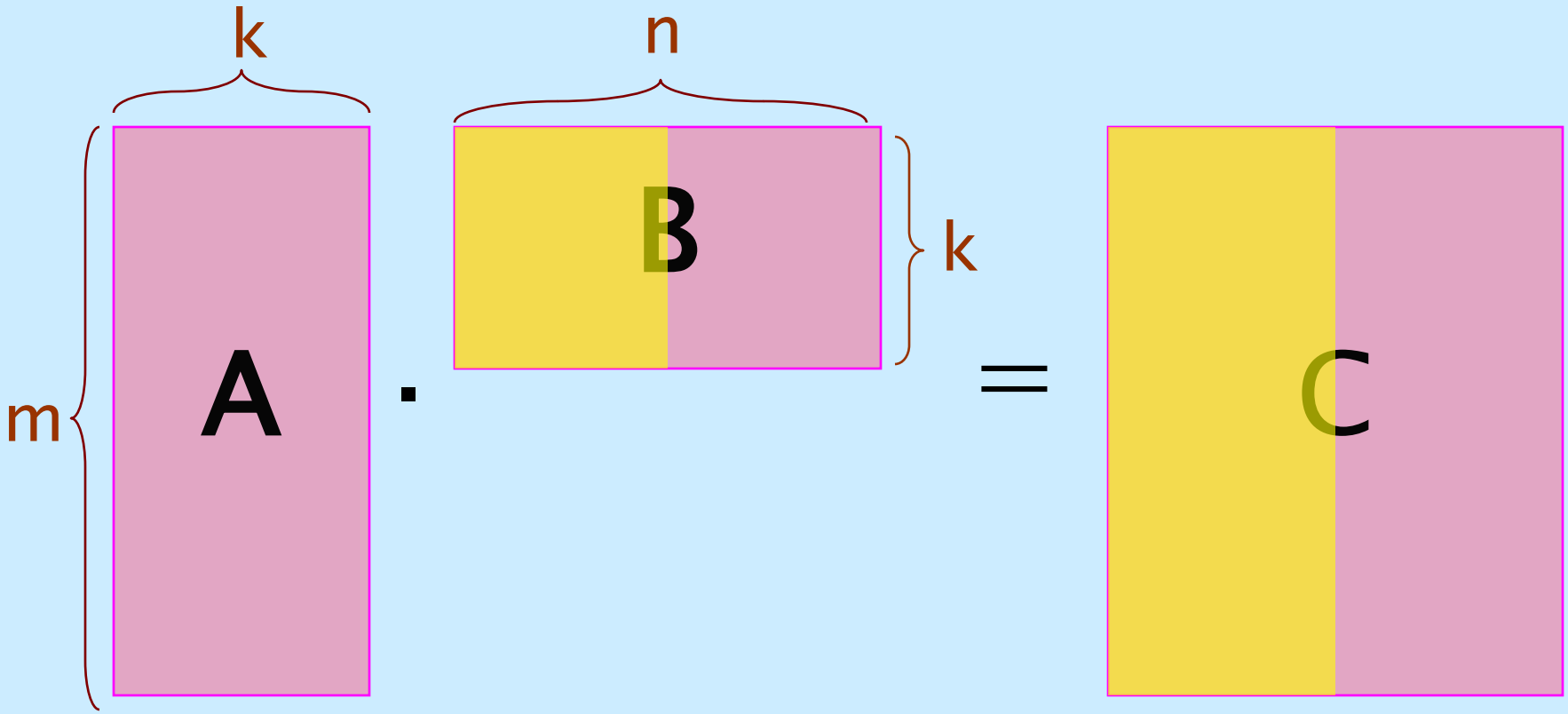
Split m

No races, safe to spawn !



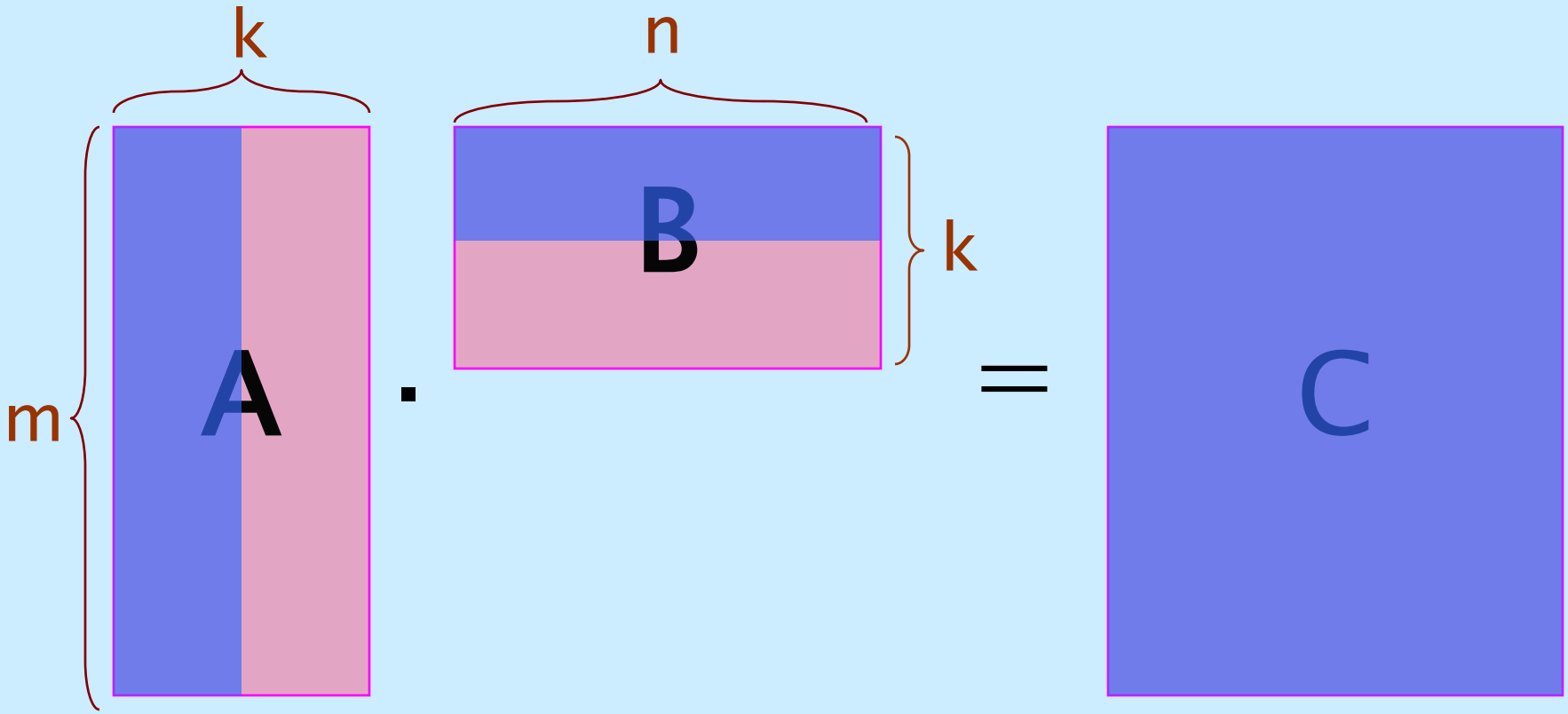
Split n

No races, safe to spawn !



Split k

Data races, unsafe to spawn !



Matrix–Vector Multiplication

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

$$y \qquad A \qquad x$$

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

Let each worker handle a single row !

Matrix-Vector Multiplication

```
template <typename T>
void MatVec (T **A, T* x, T* y, int m, int n)
{
    for(int i=0; i<m; i++) {
            for(int j=0; j<n; j++)
                    y[i] += A[i][j] * x[j];
    }
}
```


```
template <typename T>
void MatVec (T **A, T* x, T* y, int m, int n)
{
    cilk_for (int i=0; i<m; i++){
            for(int j=0; j<n; j++)
                    y[i] += A[i][j] * x[j];
   }
}
```

Matrix-Transpose x Vector

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

$$y \qquad A^T \qquad x$$

$$y_i = \sum_{j=1}^{m} a_{ji}\, x_j$$

The data is still A, no explicit transposition

Matrix-Transpose x Vector

```
template <typename T>
void MatTransVec (T **A, T* x, T* y, int m, int n)
{
    cilk_for(int i=0; i<n; i++) {
            for(int j=0; j<m; j++)
                    y[i] += A[j][i] * x[j];
    }
}
```

Terrible performance
(1 cache-miss/iteration)

Reorder loops

```
template <typename T>
void MatTransVec (T **A, T* x, T* y, int m, int n)
{
    cilk_for (int j=0; j<m; j++){
            for(int i=0; i<n; i++)
                    y[i] += A[j][i] * x[j];
    }
}
```

Data Race !

Hyperobjects

- Avoiding the data race on the variable y can be done by splitting y into multiple copies that are never accessed concurrently.
- A *hyperobject* is a Cilk++ object that shows distinct views to different observers.
- Before completing a **cilk_sync,** the parent's view is reduced into the child's view.
- For correctness, the reduce() function should be associative (not necessarily commutative).

```
template <typename T>
struct add_monoid: cilk:: monoid_base<T> {
        void reduce (T * left, T * right) const {
                *left += *right;
        }
        …
}
```

Hyperobject solution

- Use a built-in hyperobject (there are many, read the REDUCERS chapter from the programmer's guide)

```
template <typename T>
void MatTransVec (T **A, T* x, T* y, int m, int n)
{
    array_reducer_t  art(n, y);
    cilk::hyperobject<array_reducer_t> rvec(art);
    cilk_for (int j=0; j<m; j++){
            T * array = rvec().array;
             for(int i=0; i<n; i++)
                        array[i] += A[j][i] * x[j];
   }
}
```

- Use hyperobjects sparingly, on infrequently accessed global variable.
- This example is for educational purposes. There are better ways of parallelizing $y=A^Tx$.