*

Matrix Multiplication

*Material based on Chapter 10, Numerical Algorithms, of B. Wilkinson et al., "PARALLEL PROGRAMMING. Techniques and Applications Using Networked Workstations and Parallel Computers"

©2002-2004 R. Leduc

Matrix Review

A matrix is a two dimensional array of numbers.

An $n \times m$ matrix has n rows, and m columns. See Figure 10.1.

To add matrices **A** and **B** (same dimensions) gives:

$\mathbf{C} = \mathbf{A} + \mathbf{B}$ where $c_{i,j} = a_{i,j} + b_{i,j}$ $(0 \leq i < n, 0 \leq j < m)$

Multiplying two matrices **A** $(n \times l)$ and **B**$(l \times m)$ produces a matrix **C** $(n \times m)$ defined as:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} \text{ where } c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} + b_{k,j} \; (0 \leq i < n, 0 \leq j < m)$$

See Figure 10.2.

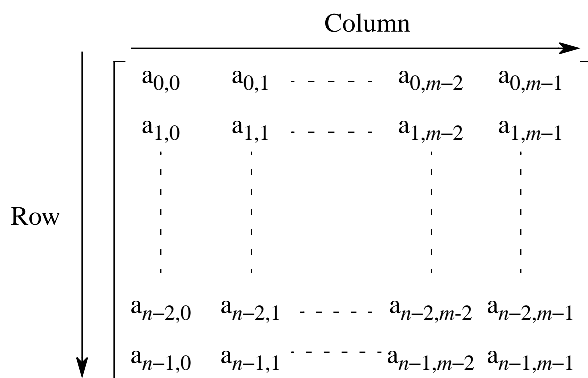

Figure 10.1 An $n \times m$ matrix.

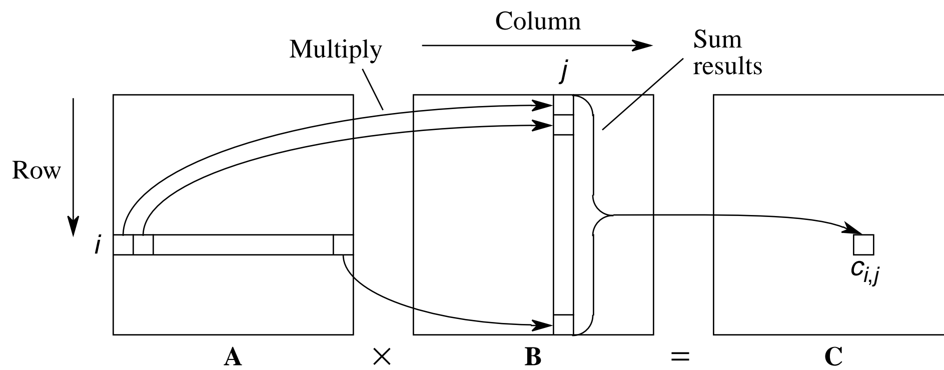

Figure 10.2 Matrix multiplication, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Sequential Matrix Multiplication Algorithm

We will assume that we are always dealing with square $n \times n$ matrices.

```
for (i = 0; i < n ; i++)
  for (j = 0; j < n; j++) {
    sum = 0;
    for (k = 0; k < n; k++)
      sum = sum + a[i][k] * b[k][j];

    c[i][j] = sum;
  }
```

Requires n^3 additions and n^3 multiplications and has time complexity $\mathbf{O}(n^3)$.

©2002-2004 R. Leduc

Parallelizing the Algorithm

Easy to see that each iteration of two outer loops are independent. ie the calculation of every `c[i][j]` can be parallelized.

For computation, if we have n^2 processors, each can evaluate a single `c[i][j]` element in $2n$ steps thus $\mathbf{O}(n)$.

This is cost optimal (see chapter 2) as:

$$\text{Cost} \quad = 2n \times n^2 = 2n^3 = t_s$$

Could even parallelize the computation of the inner loop.

©2002-2004 R. Leduc

Partitioning into Submatrices

Typically we want to use much less than n processors with an $n \times n$ matrix due to its large size.

To partition the matrix, we divide it into blocks called *submatrices.*

We can then treat these blocks as if they were matrix elements.

We will divide a matrix $\mathbf{A}$ into s^2 submatrices. Creates an $s \times s$ matrix called $\mathbf{A}_s$ whose "elements" are submatrixes of size $m \times m$, where $m = n/s$.

Use notation $\mathbf{A}_{p,q}$ to be the submatrix at row p and column q of $\mathbf{A}_s$.

©2002-2004 R. Leduc

Multiplication Algorithm Using Submatrices

For our matrix $\mathbf{A}_s$, the sequential algorithm becomes:

```
for (p = 0; p < s ; p++)
  for (q = 0; q < s; q++) {
    Cp,q = 0;              /* set to m x m Zero matrix */
    for (r = 0; r < s; r++) /* submatrix multiplication */
      Cp,q = Cp,q + Ap,r * Br,q; /* and add to */
  }                            /* accumulating submatrix */
```

In inner loop, we are multiplying two submatrices, and adding it to submatrix $\mathbf{C}_{p,q}$.

This means the inner loop would actually be composed of additional loops to do the matrix multiplication and matrix addition.

Approach is called *block matrix multiplication.* See Figures 10.4, and 10.5

©2002-2004 R. Leduc

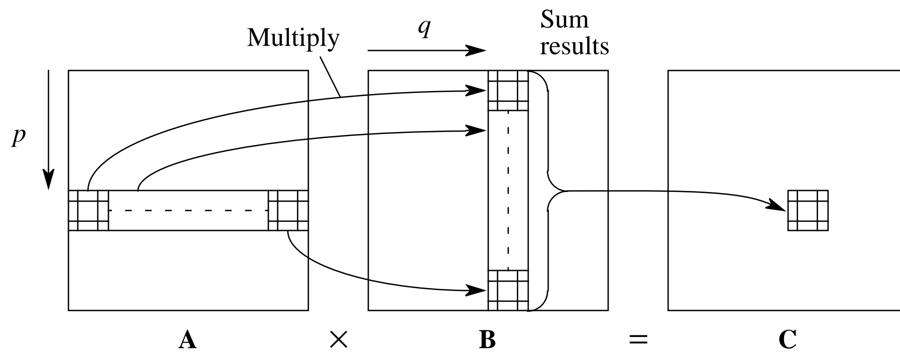

Figure 10.4 Block matrix multiplication.

$$\left[\begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right] \times \left[\begin{array}{cc|cc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array}\right]$$

(a) Matrices

$$\overset{A_{0,0}}{\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}} \times \overset{B_{0,0}}{\begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix}} + \overset{A_{0,1}}{\begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix}} \times \overset{B_{1,0}}{\begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

(b) Multiplying $A_{0,0} \times B_{0,0}$ to obtain $C_{0,0}$

Figure 10.5 Submatrix multiplication.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Direct Implementation

Will start with the simplest way to parallelize matrix multiplication.

If using n^2 processors, and each will calculate one element of matrix $\mathbf{C}$.

Means each processor needs one row of elements from $\mathbf{A}$ and one column of elements from $\mathbf{B}$.

Typically, a given row or column is sent to more than one process.

ie. to compute elements $c_{0,0}$ and $c_{0,1}$ both require row 0 of matrix $\mathbf{A}$.

If using submatrices, then use s^2 processors and each calculates an $m \times m$ submatrix of $\mathbf{C}$.

Analysis for Element-wise Algorithm

Assuming $n \times n$ matrices.

Communication: if send separate messages to each n^2 processor, must send $2n$ elements.

Each slave will return 1 element of $\mathbf{C}$.

$$\begin{aligned} t_{\text{comm}} &= n^2(t_{\text{startup}} + 2nt_{\text{data}}) + n^2(t_{\text{startup}} + t_{\text{data}}) \\ &= n^2(2t_{\text{startup}} + (2n+1)t_{\text{data}}) \end{aligned}$$

If use broadcast, we could get:

$$t_{\text{comm}} = (t_{\text{startup}} + n^2 t_{\text{data}}) + n^2(t_{\text{startup}} + t_{\text{data}})$$

Makes the dominant time the return of results as t_{startup} typically much larger than t_{data}.

©2002-2004 R. Leduc

Analysis for Element-wise Algorithm Cont.

Computation: Each slave in parallel performs n multiplications and n additions.

$$t_{\text{comp}} = 2n$$

Combining gives at best:

$$t_p = (t_{\text{startup}} + n^2 t_{\text{data}}) + n^2(t_{\text{startup}} + t_{\text{data}}) + 2n$$

Versus: $t_s = 2n^3$

Analysis for Submatrices Algorithm

To reduce the number of processors required, we will use s^2 submatrices, and thus s^2 processors.

Communication: each processor must receive one row and one column of submatrices ($2s$ submatrices in total), each with m^2 elements ($m = n/s$).

Each processor must return a $\mathbf{C}$ submatrix (m^2 elements) to the master process.

$$\begin{aligned} t_{\text{comm}} &= s^2\{(t_{\text{startup}} + 2sm^2 t_{\text{data}}) + (t_{\text{startup}} + m^2 t_{\text{data}})\} \\ &= (n/m)^2\{(t_{\text{startup}} + 2nmt_{\text{data}}) + (t_{\text{startup}} + m^2 t_{\text{data}})\} \\ &= 2(n/m)^2 t_{\text{startup}} + (n^2 + 2n^3/m)t_{\text{data}} \end{aligned}$$

If use broadcast, we could get:

$$t_{\text{comm}} = (t_{\text{startup}} + n^2 t_{\text{data}}) + (n/m)^2(t_{\text{startup}} + m^2 t_{\text{data}})$$

Analysis for Submatrices Algorithm Cont.

Computation: Each slave must process a row and a column of $m \times m$ submatrices.

Means s passes through loop that does one matrix multiplication ($2m^3$ steps) and one matrix addition (m^2 steps).

$$t_{\text{comp}} = s(2m^3 + m^2) \qquad \mathbf{O}(sm^3) = \mathbf{O}(nm^2)$$

Combining gives at best:

$$t_p = (t_{\text{startup}} + n^2 t_{\text{data}}) + (n/m)^2 (t_{\text{startup}} + m^2 t_{\text{data}}) + 2nm^2 + m$$

Versus: $t_s = 2n^3$

Recursive Implementation

In the original element-wise matrix multiplication algorithm the inner loop was: $sum = sum + a[i][k] * b[k][j]$;

The submatrix inner loop was: $Cp,q = Cp,q + Ap,r * Br,q$;

Which suggests if we continued dividing the submatrices further into smaller submatrices, we'd eventually get 1×1 matrices, assuming that n was a power of two.

At this point, the submatrix multiplication approach reduces to the element-wise algorithm.

©2002-2004 R. Leduc

Recursive Implementation Cont.

To add matrices $\mathbf{A}$ and $\mathbf{B}$, each matrix is divided into four square matrices: $A_{pp}, A_{pq}, A_{qp}, A_{qq}$ and $B_{pp}, B_{pq}, B_{qp}, B_{qq}$.

We can then recursively divide each submatrix into 4 submatrices in order to perform the submatrix multiplications, stopping when the new submatrices are single elements.

```
mat_mult(A, B,s ) {
  if (s == 1)
    C = A*B;
  else  {
    s = s/2;
    P0 = mat_mult(App, Bpp, s);
    P1 = mat_mult(Apq, Bqp, s);
    P2 = mat_mult(App, Bpq, s);
    P3 = mat_mult(Apq, Bqq, s);
    P4 = mat_mult(Aqp, Bpp, s);
    P5 = mat_mult(Aqq, Bqp, s);
    P6 = mat_mult(Aqp, Bpq, s);
    P7 = mat_mult(Aqq, Bqq, s);
    Cpp = P0 + P1;
    Cpq + P2 + P3;
    Cqp = P4 + P5;
    Cqq = P6 + P7;
  }
  return (C);
}
```

Discussion of Recursive Method

Would not want to run the recursive method directly (too much overhead).

As we saw in the divide and conquer chapter, we can use the tree construction to map the recursive problem into a parallel D & C implementation.

This would be similar to Figures 4.3 and 4.4, except using an octtree instead of a binary tree.

Normally, limit the level of recursion to match the number of processes. If had 64 processors, then would stop after the second level of recursion with $s = n/4$.

Works well for shared memory multiprocessors with local cache memory as at each stage, the size of data is reduced and localized.

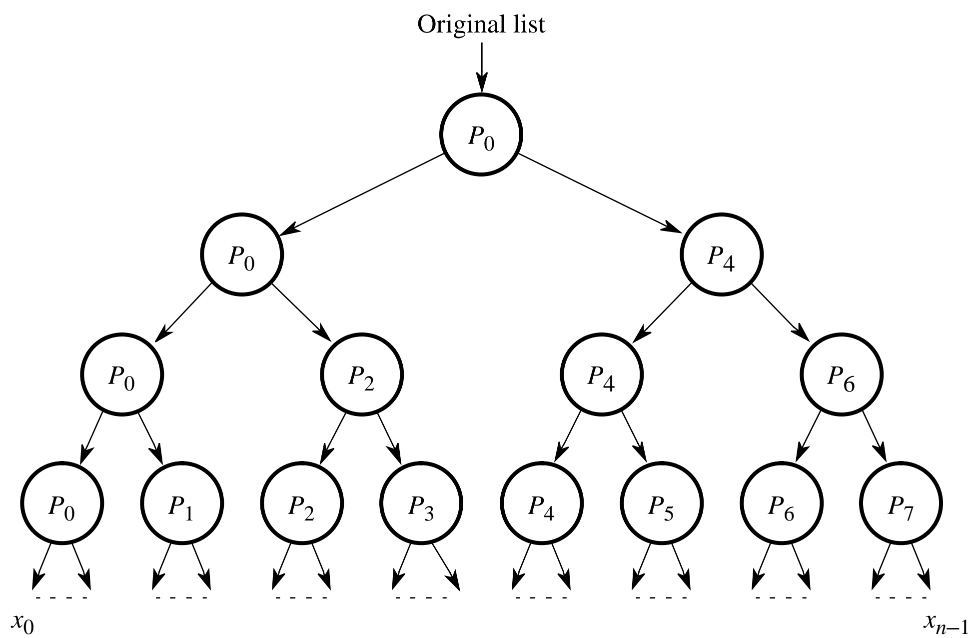

Figure 4.3 Dividing a list into parts.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

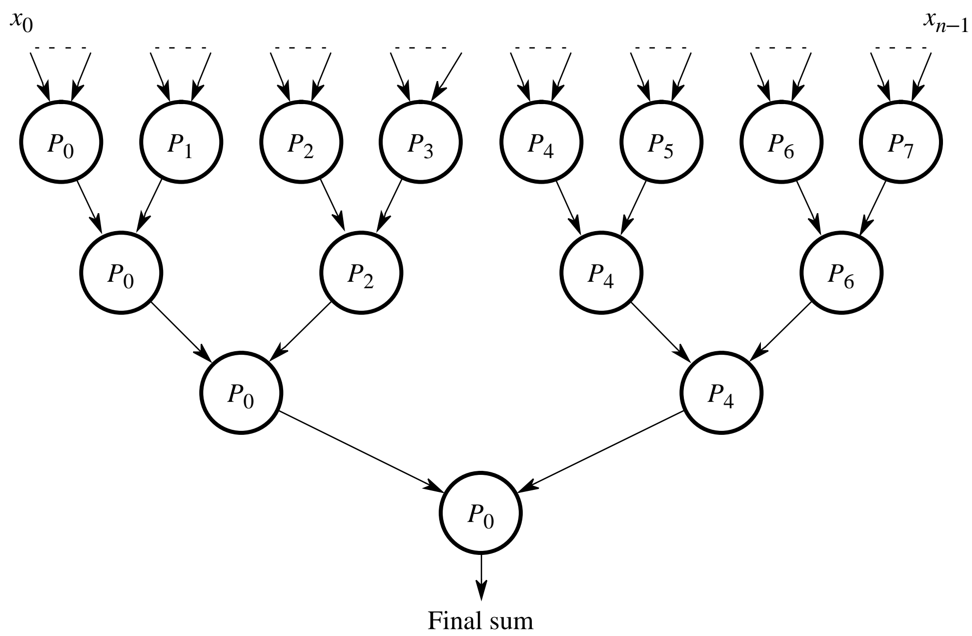

Figure 4.4 Partial summation.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Mesh Implementation

As a matrix is a two-dimensional array, a 2d mesh/torus is a natural message passing topology to use.

Allows for simultaneously shifting columns vertically and rows horizontally.

Will discuss a method called Cannon's algorithm that requires a torus structure as shifts will include wraparound.

Will disuss in terms of elements, but in reality would use submatrices.

Cannon's Algorithm

The algorithm to multiply the two $n \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$ to create $\mathbf{C}$ is:

1) At startup, each processor $P_{i,j}$ has elements $a_{i,j}$ and $b_{i,j}$ ($0 \leq i < n, 0 \leq j < n$).

2) Need to shift elements to align them to be multiplied. The entire i^{th} row of $\mathbf{A}$ is shifted i places to the left and the entire j^{th} column of $\mathbf{B}$ is shifted j places upwards. See Figure 10.10.

This puts elements $a_{i,j+i}$ and $b_{i+j,j}$ in process $P_{i,j}$, and we are ready to calculate $c_{i,j}$.

3) Each processor $P_{i,j}$ multiplies together its two elements.

©2002-2004 R. Leduc

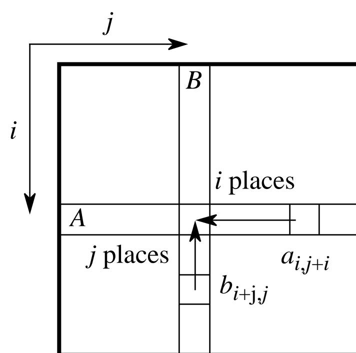

Figure 10.10 Step 2 — Alignment of elements of A and B.

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen © Prentice Hall, 1998

Cannon's Algorithm Cont.

4) The ith row of $\mathbf{A}$ is then shifted one place to the left, and the jth column of $\mathbf{B}$ is shifted 1 place upwards. See Figure 10.11.

5) Each processor $P_{i,j}$ multiplies the two new elements it receives and adds the result to its accumulating sum.

6) Steps 4 and 5 are repeated until $c_{i,j}$ calculation complete. Will perform step 4, $(n-1)$ times in total.

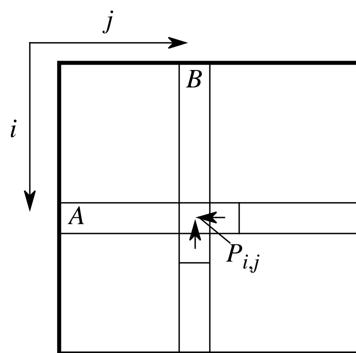

Figure 10.11 Step 4 — One-place shift of elements of *A* and *B*.

The Need to "Align" Elements

In step 2 of the algorithm, the ith row of $\mathbf{A}$ is shifted left i places, and j^{th} column of $\mathbf{B}$ is shifted j places upwards.

This is done to set up for the algorithm. To see why, look at Figure 10.5 and consider the calculation for element $c_{1,0}$ (process $P_{1,0}$).

According to step 1, $P_{1,0}$ initially contains elements $a_{1,0}$ and $b_{1,0}$. We want to calculate:

$$c_{1,0} = a_{1,0}b_{0,0} + a_{1,1}b_{1,0} + a_{1,2}b_{2,0} + a_{1,3}b_{3,0}$$

Clearly, $a_{1,0}$ and $b_{1,0}$ are not a matching pair. How ever, shifting one place left, and 0 places up, gives $a_{1,1}$ and $b_{1,0}$ which match.

From this point on, 1 left shift, and 1 up shift will create matching pairs.

Could align in different ways, but would be same amount of work.

©2002-2004 R. Leduc

Cannon's Algorithm: Analysis

We will consider Cannon's algorithm using s^2 submatrices, each $m \times m$ ($m = n/s$).

Communication: Initial alignment requires maximum of $s-1$ shifts, each requiring m^2 elements to be sent. Then $s-1$ shifts to multiply and add the matrices.

$$t_{\text{comm}} = 2(s-1)(t_{\text{startup}} + m^2 t_{\text{data}})$$

Complexity is thus $\mathbf{O}(sm^2) = \mathbf{O}(nm)$

Computation: Each processor is essentially processing a loop s times where it multiplies two submatrices together ($2m^3$ operations), and performs a matrix addition (m^2 operations) for the accumulation.

$$t_{\text{comp}} = s(2m^3 + m^2) \quad \mathbf{O}(sm^3) = \mathbf{O}(nm^2)$$

$$t_p = 2(s-1)(t_{\text{startup}} + m^2 t_{\text{data}}) + s(2m^3 + m^2)$$